



# Le C

## en 20 heures

Framabook

Eric Berthomier, Daniel Schang

# Le C en 20 heures



**La collection Framabook est un projet Framasoft**
(http ://www.framasoft.net)









Immeuble ACCET
4, place de la Pergola
95021 Cergy-Pontoise Cedex

Conception graphique de la couverture : Nadège Dauvergne

Mise en page : *La Poule ou l'Oeuf* (http ://www.lescomplexes.com/)



ISBN : 978-2-35922-030-8
Dépôt légal : 2e semestre 2010

# Avant de commencer

L'ouvrage que vous tenez dans les mains ou que vous consultez sur votre écran a pour objectif de vous faire découvrir, par la pratique, la programmation en langage C.

Il a été testé par de nombreux étudiants qui n'avaient aucune connaissance préalable de ce langage. En 20 à 30 heures de travail, ils sont tous parvenus au terme de leur apprentissage. Si vous ne connaissez encore rien à la programmation et que vous désirez apprendre, vous serez donc probablement très intéressé(e) par le contenu de cet ouvrage : il est très facile d'accès et destiné aux grands débutants.

Il existe une multitude de façons de programmer un ordinateur, qui dépendent du matériel dont vous disposez, du système d'exploitation que vous utilisez et du langage de programmation que vous choisirez. Nous avons fait le choix d'un système d'exploitation libre : Gnu/Linux et du langage C, très répandu, largement enseigné, et finalement assez simple dans ses constructions. Néanmoins, même si vous n'utilisez pas Gnu/Linux, vous pouvez sans risque vous lancer dans la lecture de cet ouvrage. Plus de quatre-vingt-quinze pour cent de ce vous y trouverez est utilisable sans modification avec d'autres systèmes d'exploitation [1].

1. Les auteurs vous encouragent néanmoins très vivement à franchir le pas, et dans le cas où vous ne voudriez pas supprimer tout simplement votre vieux système d'exploitation, rien ne vous empêche d'en avoir plusieurs sur le même ordinateur.

Ce livre n'est pas un ouvrage de référence, que vous garderez sur une étagère pour vous y reporter en cas de doute ou d'oubli. Il a été écrit pour être lu d'un bout à l'autre, dans l'ordre : il vous guidera dans votre apprentissage et vous suggèrera de programmer telle chose, de tester telle autre. En ce sens, il est orienté vers la pratique de la programmation et l'enseigne sans doute à la manière dont les auteurs l'ont apprise : devant un ordinateur, à essayer de programmer quelque chose. Vous ne pourrez donc pas profiter pleinement de cet ouvrage sans essayer de faire les nombreux exercices qu'il contient. Et lorsque vous aurez fait ces exercices, vous pourrez comparer vos solutions avec celles indiquées à la fin de chaque chapitre : vous apprendrez en écrivant du code, et en lisant du code. Vous pourrez aussi travailler à votre vitesse. Vous irez peut être vite au début et vous trouverez tout ceci très facile. Il sera néanmoins nécessaire de prendre le temps de ne pas aller trop vite : c'est pourquoi nous vous encourageons à ne pas nécessairement faire des copier/coller du code, mais à le saisir à nouveau, afin de l'assimiler, et aussi de commettre des erreurs que vous devrez ensuite corriger.

Les premières briques de cet ouvrage ont pour origine un cours de Turbo Pascal [1] qu'Éric Berthomier dispensait au sein de l'association Fac Info à l'Université de Poitiers. La seconde rangée de briques fut posée avec l'association Les Mulots à Chasseneuil du Poitou où Eric donna des cours bénévoles de C sous Turbo C 2.0 et MS/DOS. Grâce à cette association, Éric rencontra le GULP (Groupement des Utilisateurs de Linux de Poitiers) qui lui fit découvrir GNU/Linux : la troisième rangée de briques pouvait commencer. Accompagné par d'autres membres du GULP, Éric donna des cours de C au sein de cette association à l'Espace Mendès France de Poitiers.

Le contenu de l'ouvrage alors disponible sous forme de fichiers Postscript a stagné quelques années avant d'être récupéré et adapté par Daniel Schang, qui l'a utilisé et enrichi d'une quatrième rangée de briques dans un cadre plus académique à l'ESEO d'Angers.

Il ne nous sera pas possible de dire combien de versions de ce cours ont existé mais là n'est pas le plus important, ce qui compte c'est que vous ayez maintenant ce livre entre les mains et ceci grâce à l'association FramaSoft.

1. Dont quelques lignes directrices avaient elles mêmes été définies par une autre personne.

# Chapitre 1

# Premiers pas

## 1.1 Système d'exploitation et C

Pour pouvoir réaliser des programmes en C, il est nécessaire de s'appuyer sur un système d'exploitation. Le système d'exploitation utilisé est ici Gnu/Linux. Néanmoins, la quasi-totalité de ce qui est décrit ici peut être réalisé en utilisant d'autres systèmes d'exploitation.

Cet ouvrage n'est pas une introduction à l'utilisation de Gnu/Linux. Nous n'évoquerons donc que les outils nécessaires à la programmation en C.

## 1.2 Utiliser un éditeur sous Gnu/Linux

Nous allons dans cette section tenter de définir une manipulation pour lancer un éditeur [1]. Il existe une multitude d'éditeurs de texte qui permettent de saisir des programmes : Emacs (que j'utilise en ce moment même), Kate, Bluefish, Gedit…

Souvent, les éditeurs sont accessibles quelque part dans un menu. En ce qui nous concerne, nous allons lancer l'éditeur de texte *depuis la ligne de commande du shell*. Pour cela, nous allons

1. Un éditeur est un programme (comme le bloc-notes de Windows) qui nous servira à écrire nos programmes.

exécuter un *terminal*. Selon la distribution que vous utilisez le terminal n'est pas forcément rangé au même endroit. Voici quelques pistes qui pourront vous aider :

- sous Ubuntu, faites Applications / Accessoires / Terminal ;
- sous Xfce (et avec une Debian), faites clic droit, puis Terminal ;
- dans un autre environnement, recherchez quelque chose qui pourrait s'appeler *Terminal*, *Console* ou *Xterm*.

Dans cette nouvelle fenêtre qui ressemble à celle de la figure 1.1, exécutez l'éditeur Scite en tapant la commande `scite` puis en validant. L'éditeur doit s'ouvrir (voir figure 1.2).

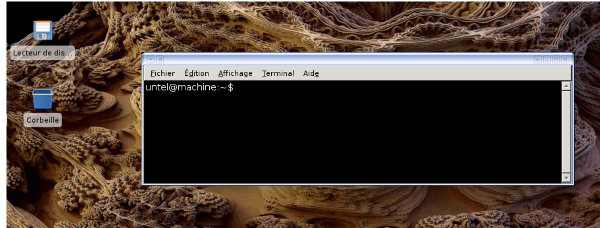



FIGURE 1.1 - Une fenêtre de terminal

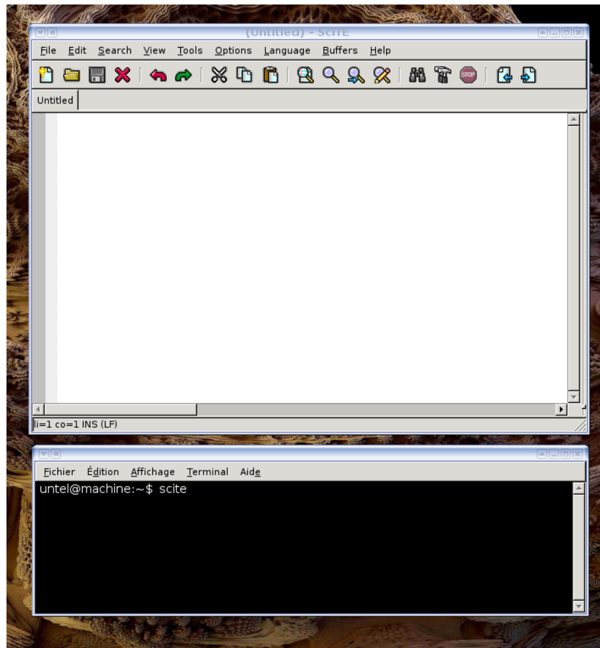



FIGURE 1.2 - Une fenêtre de terminal et l'éditeur Scite

Scite est un éditeur très simple d'utilisation. Il est de plus disponible pour plusieurs systèmes d'exploitation, est léger, peut être personnalisé, offre la coloration syntaxique, permet de « plier » les fonctions…

Si en essayant de lancer la commande `scite` vous obtenez un message d'erreur comme : Commande inconnue ou *Command not found*, c'est que Scite n'est probablement pas installé sur votre machine. Vous devrez alors regarder du côté des outils de gestion des paquetages pour ajouter ce logiciel (peut être disposez-vous des outils de gestion des paquetages : Synaptic, Aptitude, Rpmdrake, Gurpmi, Yast …).

## 1.3 Exemple de programme

Voici un premier programme. Il est fonctionnel, même s'il n'est pas normalisé [1]. Il affiche le mot `Bonjour` à l'écran. À l'aide de votre éditeur de texte (dans la fenêtre *Scite* donc), tapez le texte qui se trouve à l'intérieur du cadre suivant :

```
main () {
  puts ("Bonjour");
  getchar ();
}
```

Puis, sauvegardez ce fichier (raccourci clavier : CTRL + S ) sous le nom suivant : `programme1.c`

Une fois le texte du programme tapé, il faut le compiler, c'est à dire le transformer en programme exécutable (nous reviendrons sur la compilation plus tard). Nous allons donc ouvrir une seconde fenêtre dans laquelle nous allons compiler notre programme : comme tout à l'heure lancez un terminal (figure 1.1).

La compilation se fait avec le compilateur `gcc`. Tapez dans cette nouvelle fenêtre [2] :

```
gcc -o programme1 programme1.c
```

De la même façon que vous pourriez ne pas disposer de l'éditeur Scite, il se peut que vous n'ayez pas les outils de développement. Là aussi, selon votre distribution, recherchez l'outil de gestion des logiciels installés, et installez le compilateur Gcc. Il est probable que son installation induise l'installation d'autres outils de développement nécessaires (la plupart des outils d'installation de paquets gèrent les dépendances entre logiciels).

Si vous n'avez pas fait d'erreur, la ligne précédente ne provoquera aucun affichage (pas de nouvelle, bonne nouvelle…) La commande entrée vient de créer un fichier nommé `programme1`. Vous pouvez le vérifier en tapant `ls -l` (attention, tapez bien $\ell s - \ell$) qui devrait vous renvoyer quelque chose du type :

```
-rw--r-r-- 1 dschang dschang 44 2008-10-14 11:10 programme1.c
-rwxr-xr-x 1 dschang dschang 6525 2008-10-14 11:11 programme1
```

1. Nous verrons par la suite qu'un programme écrit en Langage C doit respecter certaines normes…
2. Il faut se placer dans le répertoire contenant `programme1.c`. Vous pouvez consulter le contenu du répertoire courant en entrant la commande ls (la lettre $\ell$ pas le chiffre 1). Vous pouvez vous déplacer dans un répertoire particulier en entrant cd <nom repertoire>.

En y regardant de plus près, nous pouvons voir le fichier intitulé `programme1.c` qui a été sauvegardé à 11h10 et qui contient `44` octets. En dessous se trouve le fichier `programme1` qui vient d'être créé et qui contient `6525` octets. Ce dernier fichier contient le code machine (code qui est compréhensible par la machine).

Le fichier `programme1` est un « fichier exécutable ». Le fichier `programme1.c` est un « fichier source » (source de tous les bonheurs de GNU/Linux…). Un fichier source désigne un fichier qu'un être humain peut comprendre par opposition à un exécutable que seule la machine arrive à comprendre. Il ne reste plus qu'à exécuter le programme :

```
./programme1
```

La machine affichera alors `Bonjour` et attendra que vous appuyiez sur ENTREE .



Nous reviendrons par la suite sur le fait qu'il faille taper `./programme1` et pas simplement `programme1`.

Par ailleurs, vous avez pu constater que la fin des lignes se terminait par un ; sauf la première ligne (celle qui contient le mot `main`). Nous reviendrons là-dessus… Disons pour l'instant que c'est un peu comme en français où chaque phrase se termine par un « . » sauf le titre.

## 1.4 Normalisation du programme

Jusqu'à présent, nous avons fait un peu « dans le sale ». Nous nous en rendrons compte en demandant au compilateur d'être plus bavard. Lancez la commande :

```
gcc -o programme1 programme1.c -Wall
```

Observez les insultes :

```
programme1.c:5: warning: return-type defaults to `int'
programme1.c: In function `main':
programme1.c:6: warning: implicit declaration of function `puts'
programme1.c:9: warning: control reaches end of non-void function
```

Les remarques du compilateur vous paraissent peut-être peu compréhensibles (voire offensantes) et c'est normal.

L'option de compilation `-Wall` permet de « déclencher la production de messages soulignant toute technique autorisée mais discutable », en deux mots : *à éviter*.

Nous allons donc normaliser ce programme.

Fondamentalement, le langage C n'est qu'un nombre restreint d'instructions et un ensemble de bibliothèques. Dans ces dernières, le compilateur trouve les fonctions et les applications qui lui permettent de créer un programme exécutable. C'est un peu ce que vous faites lorsque vous recherchez dans une encyclopédie pour réaliser un exposé.

Certaines bibliothèques (les plus courantes) sont incluses par défaut ce qui permet à notre programme de se compiler [1].

1. Pour être exact, c'est « l'édition de liens » qui a ici le mérite de fonctionner.

La fonction puts est stockée dans la bibliothèque standard d'entrées-sorties, incluse par défaut. Néanmoins, l'utilisation d'une bibliothèque nécessite que nous informions le compilateur de notre souhait de l'utiliser : il suffit d'ajouter `#include <fichier en-tête bibliothèque>` en début de programme [1].

Ainsi, puisque nous utilisons la fonction puts, qui est dans la librairie standard d'entrées/sorties, nous indiquerons en début de programme [2] : `#include <stdio.h>`

Un autre point à corriger est l'ajout de la ligne **return 0**. Tout programme *doit* renvoyer une valeur de retour, tout à la fin. Cette valeur de retour permet de savoir si le programme que l'on exécute s'est correctement terminé. En général 0 signifie une terminaison sans erreur. Enfin, il faut transformer la ligne `main ()` en `int main()`. Ce point sera détaillé par la suite lorsque nous parlerons des fonctions…

En rajoutant ces quelques correctifs nous obtenons donc :

```
#include <stdio.h>

int main () {
 puts ("Bonjour");
 getchar (); /* Permet d'attendre la frappe d'une touche */
 return 0;
}
```

## 1.5 Petit mot sur ce qu'est une bibliothèque

À l'instar de l'étudiant qui recherche dans des livres, nous pouvons dire que le « .h » représente l'index du livre et le « .c » le contenu du chapitre concerné.



Après avoir lu (et retenu) le contenu des fichiers .h inclus, si le compilateur rencontre l'appel à la fonction puts, il est en mesure de vérifier si la fonction figurait dans un des include. Si ce n'est pas le cas, il émettra un avertissement.

## 1.6 Un exemple de fichier en-tête

Vous trouverez ci-dessous, un extrait du fichier en-tête stdio.h. On y retrouve notamment la déclaration de puts (en dernière ligne de l'extrait) que nous venons de mentionner et la déclaration de printf que nous verrons dans les chapitres suivants. C'est assez compliqué… on y jette juste un oeil, pas plus ;)

```
/* Write formatted output to STREAM. */
extern int fprintf __P ((FILE *__restrict __stream,
      __const char *__restrict __format, ...));
/* Write formatted output to stdout. */
extern int printf __P ((__const char *__restrict __format, ...));
/* Write formatted output to S. */
extern int sprintf __P ((char *__restrict __s,
```

1. Le nom ajouté ici n'est pas exactement celui de la bibliothèque, mais celui du fichier d'en-tête (l'extension .h est mis pour le mot anglais *header* qui signifie en-tête) qui correspond à la bibliothèque.
2. stdio vient de *STanDard Input Output*.

```
     __const char *__restrict __format, ...));

/* Write formatted output to S from argument list ARG. */
extern int vfprintf __P ((FILE *__restrict __s,
      __const char *__restrict __format,
      _G_va_list __arg));
/* Write formatted output to stdout from argument list ARG. */
extern int vprintf __P ((__const char *__restrict __format,
     _G_va_list __arg));
/* Write formatted output to S from argument list ARG. */
extern int vsprintf __P ((char *__restrict __s,
      __const char *__restrict __format,
      _G_va_list __arg));

/* Write a string, followed by a newline, to stdout. */
extern int puts __P ((__const char *__s));
```

## 1.7 Compléments

Explicitons notre programme,

```
#include <stdio.h>

int main () {
 puts ("Bonjour");
 getchar (); /* Permet d'attendre la frappe d'une touche */
 return 0;
}
```

- `puts` : permet d'afficher du texte suivi d'un retour à la ligne.
- `getchar` : permet d'attendre la frappe d'une touche suivie d'une validation par la touche ENTREE , ou un simple appui sur la touche ENTREE .
- `/* Commentaire */` : met en commentaire tout le texte compris entre `/*` et `*/`[1]. On trouvera aussi `//` qui permet de mettre le reste de la ligne courante en commentaire[2].

Notre programme affiche donc `Bonjour` et attend que l'on appuie sur la touche entrée ou sur une autre touche puis la touche entrée.

## 1.8 Squelette de programme

On peut définir le squelette d'un programme C de la façon suivante :

```
/* Déclaration des fichiers d'entêtes de bibliothèques */

int main () {
 /* Déclaration des variables (cf. chapitres suivants...) */

 /* Corps du programme */
 getchar(); /* Facultatif mais permet d'attendre l'appui d'une touche */
```

1. Un commentaire est une portion de texte que le compilateur ignore mais qui peut aider la compréhension d'un lecteur humain.

2. Cette dernière façon de noter les commentaires provient du C++, mais elle est supportée par la plupart des compilateurs C.

```
  return 0; /* Aucune erreur renvoyée */
}
```

## 1.9 Blocs

La partie de programme située entre deux accolades est appelée un bloc. On conseille de prendre l'habitude de faire une tabulation [1] après l'accolade. Puis retirer cette tabulation au niveau de l'accolade fermante du bloc. Ainsi, on obtient :

```
int main () {
  Tabulation
  Tout le code est frappé à cette hauteur
}

Retrait de la tabulation
Tout le texte est maintenant frappé à cette hauteur.
```

Cette méthode permet de contrôler visuellement la fermeture des accolades et leurs correspondances [2].

## 1.10 Commentaires

Bien commenter un programme signifie qu'une personne ne connaissant pas votre code doit pouvoir le lire et le comprendre. Les commentaires sont indispensables dans tout bon programme. Ils peuvent être placés à n'importe quel endroit. Ils commencent par `/*` et se terminent par `*/` :

```
/* Commentaire */
```

Comme nous l'avons déjà mentionné, vous trouverez aussi parfois des commentaires C++ :

```
// Le reste de la ligne est un commentaire
```

## 1.11 Exercice d'application

Écrivez un programme qui :

- affiche « Salut toi, appuie sur une touche s'il te plaît » ;
- attend l'appui d'une touche ;
- affiche : « Merci d'avoir appuyé sur une touche ».

Une fois que vous serez satisfait de votre solution, vous pourrez la comparer avec la solution qui apparaît un peu plus loin.

---

1. La touche de tabulation TAB est la touche du clavier à gauche de la touche « A ». Cette touche sert à décaler le texte.
2. Sous Scite, CTRL + E permet d'identifier l'accolade associée à celle pointée.



Sous Linux, il est possible d'éviter de retaper à chaque fois les commandes :

Pour cela, il suffit d'appuyer plusieurs fois sur la flèche vers le haut ↑ , ce qui fera réapparaître les dernières commandes validées. Les flèches haut ↑ et bas ↓ permettent ainsi de circuler dans l'historique des commandes entrées.

## 1.12 Corrigé de l'exercice du chapitre

```
#include <stdio.h>

int main () {
  /* Affiche premier message */
  puts ("Salut toi, appuie sur une touche s'il te plaît");

  getchar (); /* Attend la frappe d'une touche */

  /* Affiche le second message */
  puts ("Merci d'avoir appuyé sur une touche");

  return 0;
}
```

## 1.13 À retenir

À l'issue de ce chapitre, il serait souhaitable de :

- Se souvenir que l'éditeur que l'on utilise dans cet ouvrage s'appelle `Scite` ;
- Connaître les fonctions `puts` et `getchar` qui apparaissent dans le programme suivant :

```
#include <stdio.h>

int main () {
  puts ("Bonjour");
  getchar (); /* Permet d'attendre la frappe d'une touche */
  return 0;
}
```

- Savoir compiler un code source de programme : `gcc -o programme1 programme1.c`
- Savoir exécuter un programme : `./programme1`

Chapitre

2

# Variables (1re partie)

Allez à votre rythme, l'important est de comprendre…

## 2.1 Objectif

Afin de stocker des valeurs, calculer, effectuer un traitement quelconque, il est nécessaire d'enregistrer de manière temporaire des données. Cet enregistrement nécessite la déclaration d'un lieu de la mémoire qui servira à ce stockage : une variable.

## 2.2 Affichage : la fonction printf

```
#include <stdio.h>

int main () {
```

```
  /* Affiche Coucou c'est moi à l'écran puis saute une ligne */
  printf ("Coucou c'est moi\n");

  return 0;
}
```

La fonction `printf`, tout comme `puts` vue précédemment, permet d'afficher une chaîne de caractères. Elle est cependant beaucoup plus puissante.



La syntaxe de `printf` est très complexe et pourrait à elle seule faire l'objet d'un chapitre, nous n'en verrons donc que des applications au fur et à mesure des besoins.

## 2.3 Notion de variable

Comme son nom l'indique, une variable est quelque chose qui varie. C'est vrai mais ce n'est pas suffisant. Une variable peut être considérée comme une boîte dans laquelle on met des données. Il est possible de lire le contenu de cette boîte ou d'écrire des données dans celle-ci.

La manière la plus immédiate de lire le contenu d'une variable est de simplement mentionner son nom.

La façon la plus simple d'affecter une valeur à une variable est l'opérateur d'affectation =.



Essayer d'utiliser une variable à laquelle nous n'avons encore affecté aucune valeur peut donner n'importe quel résultat (si on affiche le contenu d'une variable non initialisée par exemple, on pourra obtenir n'importe quelle valeur à l'écran).



Affecter une valeur à une variable ayant déjà une valeur revient à la modifier. En effet, une variable ne peut contenir qu'une seule chose à la fois. Si vous mettez une seconde donnée dans la variable, la précédente est effacée.

## 2.4 Déclaration d'une variable

La déclaration d'une variable se fait en utilisant la syntaxe suivante :

```
<son type> <son nom>;
```



Comme le montre le programme qui suit, il ne faut pas mettre les < et > comme cela apparaissait dans `<son type> <son nom>;`.

```
#include <stdio.h>

int main () {
  int i; /* déclare un entier de nom i */
  char c; /* déclare un caractère de nom c */
}
```

## 2.5 Application : exemples

### Premier exemple, avec des variables du type entier

Lisez le programme, ainsi que les explications associées.

```
#include <stdio.h>

int main () {
  int i;/* i : variable de type entier */
  int j;/* j : variable de type entier */

  i=22;/* i vaut 22 */
  j=i;/* on recopie la valeur de i dans j */
      /* donc j vaut aussi 22 à présent */

  printf ("i vaut %d\n", i);/* Affiche la valeur de i */
  printf ("i+j vaut %d\n", i+j);/* Affiche la valeur de i+j */

  return 0;
}
```



– `printf ("i vaut %d\n", i);` : `%d` signifie que l'on attend une valeur entière et qu'il faut l'afficher en décimal (base `10`). Le `%d` sera remplacé par la valeur de `i`. Cette ligne provoquera donc l'affichage suivant : `i vaut 22`
– `printf ("i+j vaut %d\n", i+j);` : dans ce cas, `%d` est remplacé par la valeur de l'expression `i+j`. Nous obtiendrons l'affichage suivant : `i+j vaut 44`

L'exécution complète de ce programme donne donc :

```
i vaut 22
i+j vaut 44
```

### Second exemple, avec des variables du type caractère

À nouveau, lisez le programme ainsi que les explications associées.

```
#include <stdio.h>

int main () {
  char car; /* car: variable de type caractère */
  char f; /* f: variable de type caractère */

  car='E';
  f='e';

  printf("car=%c f=%c\n",car,f);

  return 0;
}
```



`car='E'` : nous mettons dans la variable `car` la valeur (le code ASCII) du caractère E.

f='e' : nous mettons dans la variable f la valeur (le code ASCII) du caractère 'e'. Notons au passage que, f=e signifierait affecter la valeur de la variable e (qui n'existe pas) à f. En oubliant les *quotes* (guillemets simples). '...' nous aurions donc obtenu une erreur de compilation (variable inexistante).

L'exécution complète de ce programme affiche donc :

```
car=E f=e
```



En informatique, tout n'est que nombre ; nous parlons donc de la valeur de 'E' plutôt que de 'E' car c'est le code ASCII du caractère 'E' qui est affecté à la variable. Nous reviendrons sur ce point un peu plus tard.

## 2.6 Utilisation de % dans printf

À l'intérieur du premier paramètre d'un printf (appelé le format), l'emploi de « %x » signifie qu'à l'exécution, %x doit être remplacé par la valeur correspondante qui figure dans les paramètres suivants, après transformation de ce paramètre dans le type puis le format désigné par x. Nous avons à notre disposition plusieurs formats d'affichage, comme : %d, %x...

Exemple :

```
int i;
i=65;
printf ("Le caractère %d est %c",i,i);
```

nous donnera l'affichage suivant :

```
Le caractère 65 est A
```

- le %d est remplacé par la valeur numérique de i c'est à dire 65.
- le %c est remplacé par la valeur alphanumérique (ASCII) de i c'est à dire le caractère A (cf. table ASCII en annexe). Cette table est très utile car l'ordinateur ne « comprend » que des nombres. Elle donne une correspondance entre les lettres (que nous autres, êtres humains, comprenons) et leur codage par la machine. En résumé, chaque fois que nous manipulerons la lettre A, pour l'ordinateur, il s'agira de la valeur numérique 65...

## 2.7 Exercices

Dans les exercices qui suivent, vous devez utiliser ce que nous venons de voir sur les variables, les caractères, et sur printf.

**Exercice n°2.1** — *Déclarer, afficher (a)*



Déclarez des variables avec les valeurs suivantes 70, 82, 185 et 30 puis affichez le contenu de ces variables.



**Exercice n°2.2** — *Déclarer, afficher (b)*

Faites la même chose avec les caractères `c`, `o`, `u`, `C`, `O`, `U`.

Une fois que vous avez votre propre solution, comparez avec celle qui se trouve à la fin du chapitre…

## 2.8 Réutilisation d'une variable

Il est possible de réutiliser une variable autant de fois que l'on veut. La précédente valeur étant alors effacée :

`i = 3;`

`i = 5;`

`i = 7;` Maintenant, `i` contient `7`, les autres valeurs ont disparu.

`car = 'E';`

`car = 'G';`

`car = 'h';` La variable `car` contient maintenant le caractère `'h'`.

## 2.9 Caractères spéciaux

Certains caractères (« % » par exemple) nécessitent d'être précédés par le caractère \ pour pouvoir être affichés ou utilisés.



Pour afficher un `%` avec `printf`, nous écririons :

```
printf("La réduction était de 20\%") ;
```

Pour désigner le caractère *quote* `'` on écrira :

```
char car;
car = '\'';
```

En effet, le compilateur serait perdu avec une expression du type :

```
char car;
car = ''';
```



**Exercice n°2.3** — *Erreur volontaire*

Essayez d'écrire un programme contenant `car='''` et constatez l'erreur obtenue à la compilation.

## 2.10 Exercices



**Exercice n°2.4** — *Okay !*

Réalisez un programme qui permet d'obtenir l'affichage suivant :

```
C
'
e
s
t
Ok i vaut : 1
```



Pour pouvoir utiliser un caractère réservé à la syntaxe du C, on utilise le caractère \ comme préfixe à ce caractère (on dit que \ est un caractère d'échappement).

- pour obtenir un `"`, on utilise donc `\"`
- pour obtenir un `'`, on utilise donc `\'`

## 2.11 Corrigés des exercices du chapitre



**Corrigé de l'exercice n°2.1** — *Déclarer, afficher (a)*

```c
#include <stdio.h>

int main () {
  int i,a,b,c;
  i=70;
  a=82;
  b=185;
  c=30;

  printf ("i vaut %d.\n",i);
  printf ("a vaut %d.\n",a);
  printf ("b vaut %d.\n",b);
  printf ("c vaut %d.\n",c);

  return 0;
}
```



**Corrigé de l'exercice n°2.2** — *Déclarer, afficher (b)*

```c
#include <stdio.h>

int main () {
  char a,b,c,d,e,f;
  a='c';
  b='o';
  c='u';
  d='C';
  e='O';
  f='U';

  printf ("a vaut %c.\n",a);
```

```
  printf ("b vaut %c.\n",b);
  printf ("c vaut %c.\n",c);
  printf ("d vaut %c.\n",d);
  printf ("e vaut %c.\n",e);
  printf ("f vaut %c.\n",f);

  return 0;
}
```

**Corrigé de l'exercice n°2.3** — *Erreur volontaire*



```
#include <stdio.h>

int main () {
  char car;
  car='''; // erreur volontaire !!!
   return 0;
}
```

```
gcc -o essai essai.c
essai.c:5:6: error: empty character constant
essai.c: In function 'main':
essai.c:5: error: missing terminating ' character
essai.c:6: error: expected ';' before 'return'
```

**Corrigé de l'exercice n°2.4** — *Okay !*



```
#include <stdio.h>

int main () {
  char car;
  int i;
  i = 1;

  car = 'C';
  printf ("\n%c",car);
  car = '\'';
  printf ("\n%c",car);
  car = 'e';
  printf ("\n%c",car);
  car = 's';
  printf ("\n%c",car);
  car = 't';
  printf ("\n%c",car);

  printf ("\nOk i vaut : %d\n",i);

  return 0;
}
```

## 2.12 À retenir

**Exemples de types de variables :**

`char` permet de définir une variable de type caractère.

`int` permet de définir une variable de type entier.

**Exemple de programme avec variables :**

```
#include <stdio.h>

int main () {
  char caractere;
  int entier;
  caractere = 'c';
  entier = 1;

  printf ("caractere vaut : %c\n",caractere);
  printf ("entier vaut : %d\n",entier);

  return 0;
}
```

# Chapitre 3

# Variables (2e partie)

## 3.1 Objectif

Dans ce chapitre nous allons utiliser le langage C comme une simple calculatrice et faire des additions, multiplications…



Nous allons voir qu'afin de pouvoir utiliser la bibliothèque mathématique du langage C (`#include <math.h>`), il est nécessaire d'ajouter au moment de la compilation[1] : `-lm` (lisez bien $\ell m$) ; ce qui nous donne :

```
gcc -o monprog monprog.c -lm
```

1. Pour être précis, c'est plutôt l'étape d'édition de liens qui nécessite cette option, mais nous y reviendrons…

## 3.2 Exercice de mise en bouche



**Exercice n°3.1** — *Introduction à une calculatrice*

Écrivez un programme qui :

- écrit « Calculatrice : » et saute 2 lignes...
- écrit « Valeur de a : » et saute 1 ligne
- attend l'appui d'une touche
- écrit « Valeur de b : » et saute 1 ligne
- attend l'appui d'une touche
- écrit « Valeur de a + b : »

Normalement, vous n'aurez pas de soucis pour l'écrire... comparez ensuite avec la solution en fin de chapitre...

Pas à pas, nous allons maintenant réaliser notre petit programme de calculatrice.

## 3.3 Déclaration des variables



**Exercice n°3.2** — *Somme*

Complétez le programme en :

- déclarant 2 variables `a` et `b` de type `int` (entier) ;
- assignant à ces deux variables les valeurs `36` et `54` ;
- faisant afficher le résultat de la somme de `a+b` (attention, n'écrivez pas le résultat `90` dans votre programme !).

Pour faire afficher le résultat, il est possible d'utiliser la fonction `printf` en utilisant une troisième variable. Mais pour rester plus concis, nous afficherons directement de la façon suivante :

```
printf ("Valeur de a + b : %d",a+b) ;
```

`%d` sera remplacé par la valeur de l'expression `a+b`.

## 3.4 Saisie des variables

Si une calculatrice électronique se limitait à calculer la somme de deux nombres fixes, le boulier serait encore très répandu.



Pour saisir une variable [1], il est possible d'utiliser la fonction `scanf`. La fonction `scanf` s'utilise de la façon suivante :

```
scanf ("%d", &a); // saisie de la valeur a
```

1. Par « saisir », nous voulons dire que l'ordinateur va attendre que l'utilisateur entre une valeur au clavier puis qu'il appuie sur la touche entrée.

Comme pour `printf`, nous reconnaissons le `%d` pour la saisie d'un nombre entier. Le `&` devant le `a` signifie que nous allons écrire dans la variable `a`.

*Aïe...* En fait `&a` signifie « l'adresse mémoire de `a` ». La fonction `scanf` va donc écrire dans l'emplacement mémoire (la petite boîte contenant la variable) de `a`. Si nous oublions le `&`, nous écrirons chez quelqu'un d'autre, à une autre adresse. Et là ça peut faire mal, très mal... Mais ceci est une autre histoire sur laquelle nous reviendrons longuement par la suite... Pour l'instant, n'oubliez pas le `&`.

Nous allons maintenant saisir les variables `a` et `b`. Pour exemple, voici le code pour la saisie de `a`, la saisie de `b` reste à faire par vos soins à titre d'exercice.

```
/* Saisie de la valeur de a */
printf ("Valeur de a :\n");
scanf ("%d", &a);
```

### Initialiser les variables

Il est conseillé d'initialiser les variables avant de les utiliser. Au début du programme, après leur déclaration, assignez la valeur `0` à `a` et à `b`.



Une fois complété, compilé, nous allons tester votre programme.

Entrez une valeur pour `a` puis appuyez sur la touche ENTREE . Renouvelez ensuite l'opération pour donner la valeur de `b`. Vérifiez que le résultat est correct.



Pour aller plus rapidement, il est possible d'initialiser une variable en même temps que nous la déclarons. Pour cela, rien de plus simple : ajoutez à la fin de la déclaration le symbole d'affectation = suivi de la valeur d'initialisation :

```
int i = 10;
```

Votre programme devrait ressembler à ceci (lisez le programme puis la remarque importante qui se trouve en dessous) :

```
#include <stdio.h>
#include <math.h>

int main () {
  int a=0;
  int b=0;

  printf("Calculatrice :\n\n");
  printf("Valeur de a : \n");
  scanf("%d",&a);
  printf("\n");
  printf("Valeur de b : \n");
  scanf("%d",&b);

  printf("\nValeur de a+b : %d\n",a+b); /* Affichage de la somme */

  getchar ();
  return 0;
}
```



Vous pouvez tester ce programme et vous vous apercevrez que curieusement, le programme se finit alors que vous n'avez même pas eu le temps d'appuyer sur la touche ENTREE . C'est comme si le `getchar()` de la fin était purement et simplement oublié ? ! En fait, il s'agit d'une petite « sournoiserie » du Langage C ; en effet le `scanf("%d",&b)` attend que vous entriez *une* valeur au clavier.

Pour fixer les idées, supposons que vous entriez la valeur `1234`.

La chose à bien comprendre est que pour entrer cette valeur, vous appuyez également sur la touche ENTREE . La subtilité tient au fait que le `scanf("%d",&b)` « veut » juste *une* valeur entière. Il laissera donc la valeur de la touche ENTREE disponible pour la prochaine instruction qui ira lire quelque chose au clavier (en fait, l'appui sur la touche ENTREE reste disponible sur l'entrée standard) ; c'est précisément le `getchar()` qui va le récupérer et qui permettra donc la sortie du programme.

*Aïe aïe aïe, dur dur d'être programmeur.*

Notre idée simple de départ commence à se compliquer, et tout ça pour faire quelques opérations basiques…



**Exercice n°3.3** — *Initialisation*

Déclarez une troisième valeur de type `int` (pensez à l'initialiser à `0`) que nous nommerons simplement `s` comme somme. Une fois les valeurs de `a` et `b` saisies, initialisez `s` avec la valeur de `a+b`. Affichez la valeur de `s`. Nous devrions avoir les mêmes résultats qu'auparavant, bien sûr.



**Exercice n°3.4** — *Obtenir des résultats*

Réalisez deux petits programmes qui font :

– la soustraction de deux nombres ;
– la multiplication de deux nombres.

Une fois que vous avez votre solution, comparez avec la correction proposée plus loin.

## 3.5 Les types flottants

Nous allons étudier un nouveau type de données : les nombres à virgule flottante ou simplement flottants (`float`), qui permettent de représenter des nombres à virgule. Le type `float` permet de déclarer un tel nombre. Transformez les trois programmes précédents en utilisant le type `float` au lieu du type `int`. Enfin, si pour les `int`, nous utilisions le format `%d` au sein des `printf` et des `scanf`, à présent, nous allons utiliser le format `%f` pour les flottants.

Pour vous aider, voici un petit morceau de programme qui permet la saisie de la valeur de `a` et l'affiche :

```
float a;
printf("Saisie de a :");
scanf("%f",&a);
printf("\n  a vaut : %f\n",a);
```



Pour un affichage plus agréable il est possible de *fixer le nombre de chiffres après la virgule* de la façon suivante :

```
%.[nombre de chiffres après la virgule]f
```

Voici un exemple : `printf ("%.2f",a);`

**Exercice n°3.5** — *Ouah, les 4 opérations !*



Créez un quatrième programme qui réalise la division de deux nombres. Vous pourrez vous amuser à le tester avec une division par `0` ! La solution à ce petit problème sera vu dans le chapitre sur les conditions (`if`).

## 3.6 D'autres fonctions utiles

La fonction `abs` permet d'obtenir la valeur absolue d'un nombre entier. La fonction `fabsf` permet d'obtenir la valeur absolue d'un `float`.



Notez que pour utiliser ces deux fonctions mathématiques, il faut ajouter `#include <math.h>` dans le source et ajouter l'option `-lm` dans la commande de compilation.

**Exercice n°3.6** — *Absolument !*



Utilisez cette dernière fonction pour calculer la valeur absolue de (`a-b`).

**Exercice n°3.7** — *Arrondissez*



La fonction `ceilf` permet d'obtenir l'arrondi entier supérieur d'un flottant. Utilisez cette fonction pour calculer l'arrondi supérieur de (`a/b`).

## 3.7 Corrigés des exercices du chapitre

**Corrigé de l'exercice n°3.1** — *Introduction à une calculatrice*



```
#include <stdio.h>
#include <math.h>

int main () {
  printf("Calculatrice :\n\n");

  printf("Valeur de a : \n");
  getchar();

  printf("Valeur de b : \n");
  getchar();

  printf("Valeur de a+b :\n");
  return 0;
}
```



## Corrigé de l'exercice n°3.2 — *Somme*

```
#include <stdio.h>
#include <math.h>

int main () {
  int a,b;

  a=36;
  b=54;

  printf("Valeur de a+b : %d\n",a+b);

  getchar();
  return 0;
}
```



## Corrigé de l'exercice n°3.3 — *Initialisation*

```
#include <stdio.h>
#include <math.h>

int main () {
  int a,b;
  int s;

  a=0;
  b=0;

  printf("Calculatrice :\n\n");
  printf("Valeur de a : ");
  scanf("%d",&a);
  printf("\n");
  printf("Valeur de b : ");
  scanf("%d",&b);

  s=a+b;
  printf("Valeur de a+b : %d\n",s);

  getchar ();
  return 0;
}
```



## Corrigé de l'exercice n°3.4 — *Obtenir des résultats*

```
...
  int d;/* Résultat de la différence */
  int m;/* Résultat de la multiplication */

  d = a-b;
  printf("Valeur de a-b : %d\n", d);

  m = a*b;
  printf("Valeur de a*b : %d\n", m);
...
```



**Corrigé de l'exercice n°3.5** — *Ouah, les 4 opérations !*

```
...
 int d;/* Résultat de la division */

 d = a/b;
 printf("Valeur de a/b : %d\n", d);

 getchar ();
...
```



**Corrigé de l'exercice n°3.6** — *Absolument !*

Calculez la valeur absolue de `a-b`

```
float r;
...
r=fabsf(a-b);
printf("Valeur de r : %f\n",r);
```



**Corrigé de l'exercice n°3.7** — *Arrondissez*

Calculez l'arrondi de `a+b`

```
float arrondi;
...
arrondi=ceilf(a/b);/* Calcul de l'arrondi */
printf("Valeur arrondie : %f\n",arrondi);
```

*Remarque :* Il aurait été possible d'utiliser `%d` du fait que l'arrondi est un nombre entier !

## 3.8 À retenir

```
#include <stdio.h>

// ne pas oublier pour l'utilisation des fonctions mathématiques
#include <math.h>

int main () {
  float pi=3.14159;
  int i=10;// déclaration + initialisation
  int j;// déclaration seule (pas d'initialisation)

  // Attention, bien mettre %f et non pas %d
  printf ("pi vaut environ: %f",pi);

  printf("\n i vaut:%d\n",i);

  printf("\n entrez une valeur:");
  scanf("%d",&j);// Attention à ne pas oublier le &
  printf("\n vous venez d'entrer %d",j);

  return 0;
}
```

- Nous compilerons ce programme comme ceci [1] : `gcc -o programme1 programme1.c -lm`
- Nous le lancerons comme ceci : `./programme1`

1. Même si, en l'occurence nous n'utilisons pas de fonctions de la librairie mathématique.

Chapitre

# 4

# Conditions

## 4.1 Objectif

Dans ce chapitre, nous allons voir comment introduire des conditions dans nos programmes, de manière à ce que selon les circonstances, telle ou telle partie du code soit exécutée.

## 4.2 Exercice de mise en bouche

Écrivez un programme qui met en application le théorème de Pythagore pour calculer la longueur de l'hypoténuse d'un triangle rectangle.



Rappelons que dans un triangle rectangle, la longueur de l'hypoténuse (le plus grand côté) peut se calculer en appliquant la formule suivante :

$$\text{Longueur hypoténuse} = \sqrt{a^2 + b^2}$$

où $a$ et $b$ sont les longueurs des deux autres côtés.



La racine carrée s'obtient par l'utilisation de la fonction `sqrt(valeur)` contenue dans la bibliothèque de mathématiques (`#include <math.h>`)

$a^2$ peut s'obtenir par `a*a`.



**Exercice n°4.1** — *Pythagore*

1. Recherchez les variables nécessaires et déclarez-les dans le programme.
2. Faites saisir **a** au clavier.
3. Faites saisir **b** au clavier.
4. Appliquez la formule et affichez le résultat.



Rappelons que, afin de pouvoir utiliser la bibliothèque mathématique du C (`#include <math.h>`), il est nécessaire d'ajouter au moment de la compilation `-lm` (un tiret, la lettre $\ell$, la lettre $m$) ce qui nous donne :

```
gcc -o monprog monprog.c -lm
```

Une fois que vous êtes satisfait(e) de votre solution, vous pourrez comparer avec la solution qui se trouve à la fin de ce chapitre.

## 4.3 Condition : Si Alors Sinon

En français, nous pourrions dire quelque chose de ce type :

```
si (je travaille)
alors je progresserai
sinon je stagnerai
```

En se rapprochant un peu du Langage C, on traduirait (toujours en français) ce premier programme ainsi :

```
si (je travaille) {
  alors je progresserai
}
sinon {
  je stagnerai
}
```

Enfin, le programme en Langage C ressemblera à ceci :

```
if (je travaille) {
  je progresserai
}
else {
  je stagnerai
}
```

Les conditions s'expriment avec des opérateurs logiques dont nous allons expliquer tout de suite la signification et l'utilisation.

## 4.4 Opérateurs de comparaison

Ils servent à comparer deux nombres entre eux :

| Signification | Opérateur |
|---|---|
| Inférieur | < |
| Supérieur | > |
| Égal | == |
| Différent | != |
| Inférieur ou égal | <= |
| Supérieur ou égal | >= |

TABLE 4.1 - Opérateurs de comparaison

Voici un exemple de programme qui demande à l'utilisateur de saisir deux valeurs puis affiche la plus grande :

```
#include <stdio.h>

int main () {

  int  valeur1;
  int  valeur2;

  /* Saisie de valeur1 */
  printf ("Entrez une 1ere valeur : ");
  scanf ("%d",&valeur1);

  /* Saisie de valeur2 */
  printf ("Entrez 2eme valeur : ");
  scanf ("%d",&valeur2);

  if (valeur1<valeur2)
    printf("La plus grande valeur est: %d\n",valeur2);
  else
    printf("La plus grande valeur est: %d\n",valeur1);

  return 0;
}
```

## 4.5 Opérateurs logiques

Les opérateurs logiques permettent de combiner des expressions logiques.

<table>
<tr><th>Libellé</th><th>Opérateur</th></tr>
<tr><td>Et (and)</td><td>&&</td></tr>
<tr><td>Ou (or)</td><td>||</td></tr>
<tr><td>Non (not)</td><td>!</td></tr>
</table>

Table 4.2 - Opérateurs logiques

« | » se nomme en anglais un *pipe* (prononcer païpe). Des exemples suivront bientôt…

## 4.6 Vrai ou faux

La valeur *Vrai* peut être assimilée à la valeur numérique `1` ou à toute valeur non nulle.
La valeur *Faux* peut être assimilée à la valeur numérique `0`.
L'opérateur *Ou* (`||`) correspond alors à une addition :

| Ou | Vrai | Faux | + | 1 | 0 |
|---|---|---|---|---|---|
| Vrai | Vrai | Vrai | 1 | 2 | 1 |
| Faux | Vrai | Faux | 0 | 1 | 0 |

Table 4.3 - L'opérateur *ou*

L'opérateur *Et* (`&&`) correspond alors à une multiplication[1] :

| Et | Vrai | Faux | * | 1 | 0 |
|---|---|---|---|---|---|
| Vrai | Vrai | Faux | 1 | 1 | 0 |
| Faux | Faux | Faux | 0 | 0 | 0 |

Table 4.4 - L'opérateur *et*



On notera que :
!(Vrai) = Faux
!(Faux) = Vrai

1. L'analogie entre les opérateurs logiques d'une part et les opérations d'addition et de multiplication d'autre part n'est pas parfaite. Elle permet néanmoins de se faire une idée plus calculatoire du fonctionnement des opérateurs logiques.



Par exemple :

```
int i1 =1;
int i2 =0;
printf("i1 || i2 = %d",i1||i2);
   /* affichera 1 car : vrai||faux=vrai et vrai vaut 1 */
printf("i1 && i2 = %d",i1&&i2);
   /* affichera 0 car : vrai&&faux=faux et faux vaut 0 */
printf("contraire(1)=%d",!(1));
   /* affichera 0 car : !(vrai)=faux et faux vaut 0 */
```

## 4.7 Combinaison

Toutes les opérations logiques peuvent se combiner entre elles. Il faut néanmoins veiller aux différentes priorités des opérateurs et il faut que la condition dans sa totalité soit entourée de ().

Les exemples suivants montrent ce qui est possible et ce qui ne l'est pas :

| | |
|---|---|
| Correct | `if (car == 'a')` |
| Incorrect | `if car == 'a'` |
| Correct | `if ( car == 'a' \|\| car == 'A')` |
| Incorrect | `if (car == 'a') \|\| (car == 'A')` |
| Correct | `if ((c == 'a' \|\| c == 'A') && (c2 == 'G'))` |
| Incorrect | `if (c == 'a' \|\| c == 'A') && (c2 == 'G')` |

TABLE 4.5 - Opérateurs : formulations correctes et incorrectes



Vous verrez souvent ce type de code écrit :

```
if (er) {
  /* Alors faire quelque chose */
}
```

En appliquant ce qui a été vu précédemment, on en déduit que ce code signifie que

```
si (er != 0) /* si er différent de 0 */
{
  /* Alors faire quelque chose */
}
```

Ce qui donne en Langage C :

```
if (er != 0) { /* si er différent de 0 */
  /* Alors faire quelque chose */
}
```

Dans l'immédiat, préférez `if (er!=0)` à `if (er)` !!!

## 4.8 Accolades

Les accolades entourant les blocs d'instructions d'une condition peuvent être omises si le bloc n'est constitué que d'une seule instruction.



Voici une version lourde :

```
/* VERSION LOURDE : */
if (car == 'b') {
  printf ("car vaut b.");
}
else {
  printf ("car est différent de b.");
}
```

Voici une version plus légère :

```
/* VERSION LEGERE : */
if (car == 'b')
  printf ("car vaut b.");
else
  printf ("car est différent de b.");
```

Au contraire, dans l'exemple ci-dessous, il faut impérativement mettre des accolades !

```
if (car == 'b') { // il y a 2 instructions donc on met des { }
  printf ("car vaut ");// 1ère instruction
  printf(" %c",car);// 2ème instruction
}
else
  printf ("car est différent de b.");
```

## 4.9 Exercices



**Exercice n°4.2** — *Variable positive, négative ou nulle*

Faites saisir une variable de type entier et indiquez à l'utilisateur si celle-ci est strictement positive, strictement négative ou nulle. Votre code contiendra quelque chose comme ceci :

```
if (a > 0)
  printf ("Valeur positive");
else
  printf ("Valeur négative");
```



**Exercice n°4.3** — *Voyelles, consonnes*

Faites saisir une variable de type caractère et indiquez à l'utilisateur si celle-ci est une voyelle ou une consonne. On considérera que le caractère saisi est en minuscule.

Notez que si le caractère saisi est une lettre et n'est pas une voyelle, c'est nécessairement une consonne.

## 4.10 Corrections des exercices du chapitre



**Corrigé de l'exercice n°4.1** — *Pythagore*

```
#include <stdio.h>
#include <math.h>

int main () {
  float h; /* valeur de l'hypoténuse */
  float a; /* a,b les deux autres côtés */
  float b;

  /* Initialisation des variables par précaution */
  a = 0;
  b = 0;

  /* Saisie de a */
  printf ("Valeur du premier petit côté : ");
  scanf ("%f",&a);

  /* Saisie de b */
  printf ("Valeur du second petit côté : ");
  scanf ("%f",&b);

  /* Calcul de la longueur de l'hypoténuse */
  h = sqrt (a*a + b*b);

  /* Affichage du résultat */
  printf ("L'hypoténuse mesure : %.2f\n",h);

  /* Attendre avant de sortir */
  getchar ();

  return 0;
}
```



**Corrigé de l'exercice n°4.2** — *Variable positive, négative ou nulle*

```
#include <stdio.h>

int main () {
  /* Variable pour stocker la valeur saisie */
  int a = 0;

  /* Saisie de a */
  printf("Saisie de a : ");
  scanf("%d",&a);

  /* Strictement négative ? */
  if (a < 0)
    printf("la variable a est négative.\n");
```

```
  else {
    /* Strictement positive ? */
    if (a > 0)
      printf("la variable a est positive\n");
    /* Sinon a est nulle */
    else
      printf("la variable a est nulle\n");
  }

  getchar ();
  return 0;
}
```



**Corrigé de l'exercice n°4.3** — *Voyelles, consonnes*

```
#include <stdio.h>

int main () {
  /* Variable pour stocker la valeur saisie */
  char car;

  /* Saisie du caractère a */
  printf("Saisie du caractère : ");
  scanf("%c",&car);

  /* Test condition car voyelle minuscule */
  if ((car == 'a') || (car == 'e') || (car == 'i') || (car == 'o') || (car →
     ↪  =='u') || (car == 'y'))
    printf("la variable car est une voyelle.\n");
  else
    printf("la variable car est une consonne.\n");

  getchar ();
  return 0;
}
```

## 4.11 À retenir

- La valeur *Vrai* peut être assimilée à la valeur numérique `1` ou à toute valeur non nulle.
- La valeur *Faux* peut être assimilée à la valeur numérique `0`.
- Ne pas oublier les parenthèses lorsqu'il y a un `if` :

```
if a > 0  // ne sera pas compilé !!!
  printf ("Valeur positive");
else
  printf ("Valeur négative");
```

- au contraire, il faudrait écrire :

```
if (a > 0)
  printf ("Valeur positive");
else
  printf ("Valeur négative");
```

- Différencier l'opérateur d'affectation `=`
- Et l'opérateur de comparaison `==`.

Chapitre

# 5

# Mise au point

## 5.1 Objectif

L'objet de ce chapitre est de réaliser une pause dans l'apprentissage du C et de s'attacher à ce que l'on est capable de réaliser avec le peu de moyens que l'on a.

Ce chapitre est constitué de trois exercices de difficulté croissante. Ils nous permettront d'apprendre à utiliser une nouvelle fonction et de réaliser un exercice complet de programmation.

## 5.2 Plus petit ou plus grand



**Exercice n°5.1** — *Plus grand ou plus petit que...*

Réalisez un programme qui saisit un nombre puis indique à l'utilisateur si ce nombre est plus grand ou plus petit qu'un autre nombre défini à l'avance dans le programme.



Par exemple :

```
si (nbre_saisi < 10)
alors écrire "plus petit"
```



Vous pouvez reprendre l'exercice du chapitre sur les conditions qui indiquait si un nombre est strictement positif, strictement négatif ou nul.

## 5.3 Retour sur getchar()

La fonction `getchar()` permet d'attendre la frappe d'un caractère au clavier, de le lire et de le renvoyer. Deux utilisations peuvent donc être faites de `getchar()`,

- la première est celle permettant d'attendre la frappe d'une touche sans se soucier de sa valeur, simplement pour marquer une pause.
- la seconde est celle permettant de lire un caractère au clavier.



```
getchar();
```

```
char car;
car = getchar();
```

À chaque fois, `getchar()` effectue le même traitement :

- attendre la frappe d'une touche au clavier suivie de la touche ENTREE
- renvoyer le caractère frappé.

Dans le premier exemple, ce caractère est simplement ignoré.

## 5.4 Boucle : Faire … Tant que (condition)

`do … while` (traduisez par *Faire … Tant que*) permet de réaliser une suite d'instructions tant qu'une ou plusieurs conditions sont remplies.



1. lisez le programme suivant
2. **lisez les explications qui figurent en dessous du programme**
3. exécutez, testez, comprenez ce programme…

```
#include <stdio.h>

int main () {
  char car=' ';
  int sortie=0;
  do {
    printf ("Appuyez sur S pour sortir !\n");
```

```
    car = getchar ();
    /* On le compare pour savoir si l'on peut sortir: */
    sortie = ((car == 's') || (car == 'S'));
  }
  while (sortie==0);
  return 0;
}
```



Notez que la touche ENTREE utilisée pour la saisie du caractère est elle-même traitée comme un caractère entré et provoque donc aussi l'affichage de la phrase « Appuyez sur S pour sortir... » qui s'affiche donc deux fois.



- un nombre entier vaut la valeur logique vraie si ce nombre est différent de `0`.
- un nombre entier vaut la valeur logique faux si ce nombre est égal à `0`.
- `||` est un « 'ou' logique », donc au niveau de la ligne :

  `sortie=((car=='s')||(car=='S'));`

  lorsque `car` vaudra `'s'` ou `'S'`, `sortie` vaudra `1` c'est-à-dire *Vrai*. Dans tous les autres cas, `sortie` vaudra `0`, soit la valeur *Faux*.

  Dès lors, les deux extraits de programme suivants sont équivalents :

```
sortie = ((car == 's') || (car == 'S'));
```

```
if ((car == 's') || (car == 'S'))
  sortie=1;
else
  sortie=0;
```

Pour stopper un programme qui boucle, il faut presser simultanément sur les touches CTRL + C (*break*).

Vous pouvez tester cette possibilité en exécutant le programme suivant :

```
#include <stdio.h>
int main () {
  int i=1;
  do {
    printf(" i=%d \n",i);
    i=i+1;
  }
  while (i>0); /* Test toujours vrai ! */
  return 0;
}
```



**Exercice n°5.2** — *Sortir de la boucle*

Transformez l'exemple précédent afin que l'on sorte de la boucle uniquement quand l'utilisateur a saisi le nombre 10.



La saisie d'un nombre ne se fait pas par `getchar` mais par `scanf`.

## 5.5 Opérateur modulo

L'opérateur qui donne le reste de la division entière (opérateur modulo) est noté `%` en C. Ainsi, `10%2` vaut `0` car la division entière de `10` par `2` donne `5` et il n'y a pas de reste. En revanche, `10%3` vaut `1` car la division entière de `10` par `3` donne `3` et il reste `1`.



Par exemple :

```
int z;
z=10%2;
printf("10 modulo 2=%d\n",z);
z=10%3;
printf("10 modulo 3=%d\n",z);
```

...va nous afficher :

```
10 modulo 2=0
10 modulo 3=1
```

## 5.6 Nombres pseudo-aléatoires

Voici un petit exemple de programme qui permet d'obtenir des nombres pseudo-aléatoires entre `0` et `99` :



```
#include <stdio.h>
#include <stdlib.h> // sert pour les fonctions srand et rand
#include <time.h>

int main() {
  int nb_alea=0;

  /* Initialisation du générateur de nombres
     basée sur la date et l'heure */
  srand (time (NULL));

  nb_alea = rand() % 100;
  printf ("Nombre aléatoire : %d\n",nb_alea);
  return 0;
}
```



- `srand (time (NULL))` permet d'initialiser le générateur de nombres pseudo-aléatoire. Nous reviendrons sur ce point par la suite.
- `rand()` renvoie un nombre entier compris entre `0` et `RAND_MAX`.
- `rand()%100` est donc le reste de la division entière d'un nombre pseudo-aléatoire (éventuellement très grand) par 100, c'est à dire un nombre compris entre 0 et 99...

```
#include <stdio.h>
#include <stdlib.h>

int main() {
```

```
  /* Pour notre information */
  printf ("RAND_MAX : %ld\n", RAND_MAX);

  return 0;
}
```



**Exercice n°5.3** — *Deviner un nombre*

En vous aidant de ce qui a été fait précédemment, réalisez un petit jeu qui :

1. Initialise un nombre entre `0` et `99`.
2. Tente de faire deviner ce nombre à l'utilisateur en lui indiquant si le nombre à trouver est plus petit ou plus grand que sa proposition.



Voici un exemple de dialogue avec l'ordinateur :

```
Entrez votre nombre: 50
C'est plus !

Entrez votre nombre: 25
C'est moins !

...

Gagné !!!
```

## 5.7 Corrigés des exercices du chapitre



**Corrigé de l'exercice n°5.1** — *Plus grand ou plus petit que...*

```
#include <stdio.h>

int main () {
  int nb_choisi = 33;
  int nb_saisi = 0;

  printf ("Votre nombre : ");
  scanf ("%d",&nb_saisi);

  if (nb_choisi < nb_saisi)
    printf ("Mon nombre est plus petit.\n");
  else {
    if (nb_choisi == nb_saisi)
      printf ("C'est exactement mon nombre.\n");
    else
      printf ("Mon nombre est plus grand.\n");
  }

  return 0;
}
```



## Corrigé de l'exercice n°5.2 — *Sortir de la boucle*

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int valeur=0;

  do {
    printf ("Votre nombre : ");
    scanf ("%d",&valeur);
  }
  while (valeur != 10);

  return 0;
}
```



## Corrigé de l'exercice n°5.3 — *Deviner un nombre*

```
#include <stdio.h>
#include <stdlib.h> /* pour les valeurs aléatoires */
#include <time.h>

int main () {
  int nb_hasard = 0;
  int votre_nb = 0;

  srand (time (NULL));
  nb_hasard = rand () % 100 ;/* Nombre entre 0 et 100 */

  do {
    printf("Votre nombre : ");
    scanf("%d",&votre_nb);

    if (nb_hasard < votre_nb)
      printf ("\nMon nombre est plus petit\n");
    else {
      if (nb_hasard > votre_nb)
        printf ("\nVotre nombre est plus grand\n");
    }
  }
  while (votre_nb != nb_hasard);

  printf ("Trouvé\n");

  return 0;
}
```

Chapitre

# 6

# Et les Shadoks pompaient : je pompe donc je suis

*« Les Shadoks » est une série télévisée d'animation française en 208 épisodes de 2 à 3 minutes, créée par Jacques Rouxel, produite par la société AAA (Animation Art-graphique Audiovisuel) et diffusée entre le 29 avril 1968 et 1973 (trois premières saisons) et à partir de janvier 2000 (quatrième saison) sur Canal+ et rediffusée sur Cartoon Network...*

## 6.1 Objectifs

Vous trouverez beaucoup d'exercices dans ce chapitre où plusieurs petites aides de programmation seront données notamment en ce qui concerne l'incrémentation et la décrémentation d'une variable.

## 6.2 Boucle While

De la même façon que :

`do {…} while(condition);`

il est possible d'utiliser :

`while(condition) {}`



```
char car = ' ';
while ((car != 's') && (car != 'S')) {
  car = getchar ();
}
```



Donnée seule, la ligne : `while(condition);` signifie que tant que condition est vraie, on revient à la même ligne.

Si la condition est toujours vraie, on tombe alors dans un puits (le programme reste bloqué). Sinon, on passe immédiatement à la ligne/instruction suivante.

## 6.3 Et les Shadoks apprenaient que reprendre équivaut à apprendre

*Ce n'est qu'en essayant continuellement que l'on finit par réussir… En d'autres termes… Plus ça rate et plus on a de chances que ça marche…* (http ://www.lesshadoks.com).

**Exercice n°6.1** — *Touche-touche*



Traduisez en langage C, complétez avec les variables nécessaires, compilez, exécutez, comprenez :

```
Faire
  Saisir une touche
Tant Que (touche != S) et (touche != s)
```

**Exercice n°6.2** — *Saisir un nombre*



Traduisez en langage C, complétez avec les variables nécessaires, compilez, exécutez, comprenez :

```
Faire
  Saisir un nombre
Tant Que (nombre != 10)
```



Notez que la saisie d'un nombre ne se fait pas par la fonction `getchar()` mais par la fonction `scanf` (reportez-vous à la section 3.4 pour plus de précisions).

## 6.4 Fonction toupper()

Le problème de la comparaison de la minuscule et de la majuscule de l'exercice 1 peut être contourné par l'utilisation de la fonction `toupper` qui transforme un caractère minuscule en majuscule. Pour l'utiliser, il faut inclure le fichier d'en-tête `ctype.h` par : `#include <ctype.h>`.

La fonction `toupper` s'utilise de la façon suivante :

```
#include <stdio.h>
#include <ctype.h>

int main () {
  char car;
  char car_min;

  car_min = 'a';
  car = toupper (car_min);
  printf ("%c",car);

  return 0;
}
```

Ce programme affichera : `A`

## 6.5 Ô tant qu'en emporte le Shadok



**Exercice n°6.3** — *Recommencez ! (a)*

Écrivez le programme : *Tant que je ne tape pas un nombre impair compris entre 1 et 10 je recommence la saisie d'un nombre.*



**Exercice n°6.4** — *Recommencez ! (b)*

Écrivez le programme : *Tant que je ne tape pas une voyelle je recommence la saisie d'une touche.*

## 6.6 Et les Shadoks continuaient à pomper pour obtenir le résultat

Dans les exercices qui suivent, la notion de compteur intervient. Un compteur est une variable numérique que l'on décrémente (-1) ou incrémente (+1) suivant nos besoins.

Par exemple :



```
int i;
i=1;
i=i+1;

printf("i vaut: %d",i); // affichera 2
```

En effet, lorsque nous écrivons `i=1`, l'ordinateur range la valeur 1 dans la case mémoire désignée par `i`. Lorsque nous demandons : `i=i+1`, l'ordinateur commence par prendre connaissance

de la valeur de i en mémoire (ici 1) et lui ajoute 1. Le résultat est stocké dans la case mémoire associée à i. Finalement, i vaudra 2.

Pour gagner du temps, le langage C nous permet de remplacer une expression comme : i=i+1 par l'expression suivante : i++ qui fera exactement la même chose.



```
int i;

i++; /* Incrémente le compteur i */
i--; /* Décrémente le compteur i */
```

Dans les exemples précédents, le nombre de caractères entrés peut donc être comptabilisé en ajoutant 1 à une variable à chaque fois qu'une touche est frappée.



**Exercice n°6.5** — *Recommencez ! (c)*

Écrivez le programme : *Tant que je n'ai pas saisi 10 nombres, je recommence la saisie d'un nombre.*



**Exercice n°6.6** — *Recommencez ! (d)*

Écrivez le programme : *Tant que je n'ai pas saisi 10 caractères, je recommence la saisie d'un caractère.*

Dans les exercices qui précèdent, de petites difficultés peuvent surgir… Nous vous invitons donc à vous pencher plus rapidement sur la solution.

## 6.7 Dans le clan des Shadoks, on trie, voyelles, chiffres premiers



**Exercice n°6.7** — *Recommencez ! (e)*

Écrivez le programme : *Tant que je n'ai pas saisi 10 voyelles, je recommence la saisie d'une touche.* Vous prendrez soin d'indiquer à l'utilisateur combien de voyelles il lui reste à entrer.



**Exercice n°6.8** — *Recommencez ! (f)*

Écrivez le programme : *Tant que je n'ai pas saisi 10 chiffres premiers (2,3,5 ou 7), je recommence la saisie d'un chiffre.* Vous prendrez soin d'indiquer à l'utilisateur combien de chiffres premiers il lui reste à entrer.

## 6.8 Incrémentations, pré-incrémentations…

Nous avons vu qu'incrémenter désignait la même chose qu'augmenter la valeur d'une variable de 1 :

$$\texttt{i++;} \iff \texttt{i=i+1;}$$

Il y a cependant une nuance subtile.

| | |
|---|---|
| `x=i++` | Copie d'abord la valeur de `i` dans `x` et **incrémente `i` après** |
| `x=i--` | Copie d'abord la valeur de `i` dans `x` et **décrémente `i` après** |
| `x=++i` | **Incrémente `i` d'abord** puis copie le contenu de `i` dans `x` |
| `x=--i` | **Décrémente `i` d'abord** puis copie le contenu de `i` dans `x` |

Table 6.1 - Incrémentation / Décrémentation

Pour bien comprendre, étudions le programme suivant :

```
#include <stdio.h>

int main () {
 int n=5;
 int x;

 x=n++;
 printf ("x: %d n: %d\n",x,n);

 return 0;
}
```

Celui-ci retourne le résultat suivant :

```
x: 5 n: 6
```

Analysons ce qui se passe lors de l'application de la ligne (`x=n++`) :

1. on affecte `n` à `x`, donc `x` va contenir la valeur `5`
2. on augmente `n` de `1`, donc `n` vaudra `6`

On parlera dans ce cas d'incrémentation post-fixée…

Voici un autre exemple :

```
#include <stdio.h>

int main () {
 int n=5;
 int x=0;

 x=++n;
 printf ("x: %d n: %d\n",x,n);

 return 0;
}
```

Celui-ci retourne le résultat suivant :

```
x: 6 n: 6
```

Analysons ce qui se passe lors de l'application de la ligne(`x=++n`) :

1. on augmente `n` de `1`, donc `n` vaudra `6`
2. on affecte `n` à `x`, donc `x` va contenir la valeur `6`

On parlera dans ce cas d'incrémentation pré-fixée…

## 6.9 Corrigés des exercices du chapitre



**Corrigé de l'exercice n°6.1** — *Touche-touche*

```
#include <stdio.h>

int main () {
  char car = '\0';
  printf("Tapez 's' ou 'S' pour arrêter ...\n");
  do {
    car = getchar ();
  }while ((car != 's') && (car != 'S'));
  return 0;
}
```



**Corrigé de l'exercice n°6.2** — *Saisir un nombre*

```
#include <stdio.h>

int main () {
  int nbre = 0;
  printf("Tapez 10 pour arrêter ...\n");
  do {
    scanf ("%d", &nbre);
  }while (nbre != 10);
  return 0;
}
```



**Corrigé de l'exercice n°6.3** — *Recommencez ! (a)*

```
#include <stdio.h>

int main () {
  int nbre = 0;

  printf("Tapez un chiffre impair pour arrêter ...\n");
  while ((nbre!=1) && (nbre!=3) && (nbre!=5) && (nbre!=7) && (nbre!=9))
    scanf("%d", &nbre);

  return 0;
}
```

ou bien :

```
#include <stdio.h>

int main () {
  int nbre = 0;

  printf("Tapez un chiffre impair pour arrêter ...\n");
  while ((nbre < 0) || (nbre >= 10) || (nbre%2==0))
    scanf("%d", &nbre);

  return 0;
}
```



**Corrigé de l'exercice n°6.4** — *Recommencez ! (b)*

```
#include <stdio.h>
#include <ctype.h>

int main () {
  char car = '\0';

  printf("Tapez une voyelle pour arrêter ...\n");

  while ((car != 'A') && (car != 'E') && (car != 'I') &&
    (car != 'O') && (car != 'U') && (car != 'Y')){

    car = getchar ();
    car = toupper (car);
  }
  return 0;
}
```



**Corrigé de l'exercice n°6.5** — *Recommencez ! (c) [Il conviendrait dans cet exercice de vérifier que l'utilisateur a bien entré uniquement des nombres et pas aussi des caractères. Le lecteur intéressé pourra consulter la documentation de* `scanf` *et essayer d'utiliser la valeur que renvoie cette fonction pour régler ce problème.]*

```
#include <stdio.h>

int main () {
  int nbre = 0;
  int nb_nbre = 0;

  printf("Tapez 10 nombres pour arrêter ...");

  do {
    scanf("%d",&nbre);
    nb_nbre ++;
  }while (nb_nbre != 10);

  return 0;
}
```



**Corrigé de l'exercice n°6.6** — *Recommencez ! (d)*

Prenez soin de lire l'exemple d'exécution et son explication :

```
#include <stdio.h>

int main () {
  char car = '\0';
  int nbre = 0;

  printf("Tapez 10 caractères pour arrêter ...");

  do {
    car = getchar ();

    nbre ++;
    printf("j'ai lu (%c)   nbre=%d\n",car,nbre);
  }while (nbre != 10);
```

```
  return 0;
}
```

Voici un exemple d'exécution :

```
Tapez 10 caractères pour arrêter ... 123456789
j'ai lu (1) nbre=1
j'ai lu (2) nbre=2
j'ai lu (3) nbre=3
j'ai lu (4) nbre=4
j'ai lu (5) nbre=5
j'ai lu (6) nbre=6
j'ai lu (7) nbre=7
j'ai lu (8) nbre=8
j'ai lu (9) nbre=9
j'ai lu (
)  nbre=10
```

L'utilisateur a donc tapé **123456789** suivi de la touche entrée, il a donc bien tapé 10 caractères. Ce que montre l'affichage de la dernière ligne avec la parenthèse ouvrante sur une ligne et la parenthèse fermante sur la ligne suivante.

Vous pouvez faire d'autres essais...



### Corrigé de l'exercice n°6.7 — *Recommencez ! (e)*

```
#include <stdio.h>
#include <ctype.h>

int main () {
  char car;
  int nb_nbre = 10;

  printf("Tapez encore %d voyelles pour arrêter...\n",nb_nbre);
  do {
    car=getchar();

    car=toupper(car);
    if (car=='A' || car=='E' || car=='I' || car=='O' || car=='U') {
      nb_nbre--;
      printf("Tapez encore %d voyelles pour arrêter...\n",nb_nbre);
    }
  }
  while (nb_nbre != 0);

  return 0;
}
```



### Corrigé de l'exercice n°6.8 — *Recommencez ! (f)*

```
#include <stdio.h>
#include <ctype.h>

int main () {
  int nb_nbre = 10;
  int nbre;

  printf("Tapez encore %d chiffres premiers...\n",nb_nbre);
```

```
  do {
    scanf("%d",&nbre);
    if (nbre==2 || nbre==3 || nbre==5 || nbre==7) {
      nb_nbre--;
      printf("Tapez encore %d chiffre(s) premier(s) pour arrêter...\n", →
      ↪ nb_nbre);
    }
  }
  while (nb_nbre != 0);

  return 0;
}
```

## 6.10 À retenir

Voici un exemple de programme qui résume ce qui a été vu dans ce chapitre. Ce programme doit afficher à l'écran tous les nombres pairs inférieurs à 100 :

```
#include <stdio.h>

int main () {
  int i = 0;
  while ( i!=100) {
    if (i%2==0) /* reste de la division de i par 2 */
      printf("%d ",i);
    /* pas de else ni de {} ici, c'est inutile...*/

    i++;
  }
  return 0;
}
```

Voici une autre version (meilleure) :

```
#include <stdio.h>

int main () {
  int i = 0;
  while ( i!=100) {
    printf("%d ",i);
    i+=2;
  }
  return 0;
}
```

Enfin, n'oubliez pas le tableau :

| | |
|---|---|
| x=i++ | Copie d'abord la valeur de i dans x et **incrémente i après** |
| x=i-- | Copie d'abord la valeur de i dans x et **décrémente i après** |
| x=++i | **Incrémente i d'abord** puis copie le contenu de i dans x |
| x=--i | **Décrémente i d'abord** puis copie le contenu de i dans x |

TABLE 6.2 - Incrémentation / Décrémentation (bis)

# Chapitre 7

# Boucles

## 7.1 Et les Shadoks pédalèrent pendant 15 tours

Afin d'effectuer un certain nombre de fois une tâche, nous utilisons l'instruction `for` de la façon suivante (avec `i`, une variable de type entier (`int` par exemple)) :

```
for (i=point de départ; i<point d'arrivée; i=i+pas) {
  instruction(s) répétées(s);
}
```

Ceci est rigoureusement équivalent à cela :

```
i=point de départ;

while (i<point d'arrivée) {
  instruction(s) répétées(s);

  i=i+pas;
}
```

Par souci de simplicité, nous dirons simplement que l'incantation suivante :

```
for (i=0; i<15; i++) {
 instr;
}
```

signifie que l'on va exécuter `instr` pour `i` variant de `0` à `14` inclus (`<15`) c'est à dire `15` fois.



Par exemple :

```
#include <stdio.h>
int main () {
 int i;
 for (i=0; i<15; i++){
  printf("Je me répète pour i valant %d\n",i);
 }
 printf("L'instruction a été répétée... 15 fois\n");

 return 0;
}
```

## 7.2 Syntaxe

De la même façon que le `if`, le `for` ne nécessite pas d'accolades s'il n'y a qu'une instruction à répéter.



Nous pouvons utiliser cette fonctionnalité dans le programme précédent en remplaçant :

```
for (i=0; i<15; i++) {
 printf("Je me répète pour i valant %d\n",i);
}
```

par :

```
for (i=0; i<15; i++)
 printf ("Je me répète pour i valant %d\n",i);
```

## 7.3 Notion de double boucle

Il est possible de remplacer les instructions contenues dans une boucle par une autre boucle afin de réaliser une double boucle. Nous obtenons donc :

```
Pour i allant de ... à ... {
  ...
  Pour j allant de ... à ... {
    ...
  }
}
```

Par exemple :

```
#include <stdio.h>
int main () {
  int i;
  int j;
  for (i=0; i<5; i++){
    printf("\nJe suis dans la boucle i, i vaut %d\n",i);

    for (j=3; j>0; j--)
      printf("Je suis dans la boucle j, j vaut %d\n",j);
  }

  return 0;
}
```

**Exercice n°7.1** — *Étoiles*

En utilisant une double boucle, écrire un programme qui affiche une étoile, passe à la ligne, affiche deux étoiles, passe à la ligne, affiche trois étoiles... jusqu'à cinq étoiles afin d'obtenir ceci :

```
*
**
***
****
*****
```

# 7.4 Et les Shadoks fêtèrent Noël...

## 7.4.1 Affichage des ramures du sapin

**Exercice n°7.2** — *Sapin*

À l'aide d'une double boucle, réalisez un cône pour dessiner le haut du sapin sur 10 lignes.

Vous devriez obtenir ceci :

```
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
```



Instructions :

– sur la ligne n° 1, afficher 9 espaces puis 1 étoile ;
– sur la ligne n° 2, afficher 8 espaces puis 3 étoiles ;
– sur la ligne n° 3, afficher 7 espaces puis 5 étoiles ;
– sur la ligne n° i, afficher 10-i espaces puis 2*i-1 étoiles.

Nous obtenons donc par exemple pour l'affichage des étoiles sur la ligne i :

```
for (j=0; j<((2*i)-1); j++) {
 printf ("*");
}
```

### 7.4.2 Affichage du tronc



**Exercice n°7.3** — *Tronc*

Pour poursuivre le sapin, il nous faut maintenant dessiner le tronc. Écrivez la suite du programme en dessinant le tronc à l'aide du caractère @. Vous devriez obtenir ceci (juste pour le tronc) :

```
@@@
@@@
@@@
```

## 7.5 Table Ascii

Les codes Ascii (c'est-à-dire les nombres qui représentent les caractères en informatique) vont de 0 à 255.



**Exercice n°7.4** — *Code Ascii*

Écrivez un programme qui fait afficher, sur des lignes successives, les codes ASCII avec les caractères qui leur correspondent (vous pouvez commencer l'affichage à partir du code 32 et vous arrêter au code 127).

Pour faire afficher le caractère associé à un code ASCII, vous utiliserez l'instruction ci-dessous (le %3d signifie que le nombre va être affiché en utilisant un emplacement qui fait trois caractères de long) :

```
printf ("%3d : %c", code_ascii, code_ascii);
```



Par exemple :

```
int i = 65;
printf("%3d : %c", i, i); // affichera 65 : A
```



**Exercice n°7.5** — *Un beau tableau*

Même chose, mais avec un joli affichage, par exemple sous la forme d'un tableau sur huit colonnes...

## 7.6 Corrigés des exercices du chapitre



**Corrigé de l'exercice n°7.1** — *Étoiles*

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int i;
  int j;
  for (i=1; i<=5; i++){
    for (j=1; j<=i; j++){
      printf ("*");
    }
    printf ("\n");
  }

  return 0;
}
```

Nous aurions aussi pu écrire :

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int i;
  int j;
  for (i=1; i<=5; i++){
    for (j=1; j<=i; j++) // pas d'accolades nécessaires...
      printf("*");

    printf("\n");
  }

  return 0;
}
```



**Corrigé de l'exercice n°7.2** — *Sapin*

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int i;
  int j;

  for (i=1; i<=10; i++) {
    for (j=0;j<10-i;j++) // les espaces...
      printf(" ");

    for (j=0; j<(i*2-1); j++)
      printf("*");
```

```
    printf("\n");
  }

  return 0;
}
```



## Corrigé de l'exercice n°7.3 — *Tronc*

Vous ajoutez ceci :

```
  ...
  for (i=1; i<=3; i++)
     printf("  @@@\n");
  ...
```



## Corrigé de l'exercice n°7.4 — *Code Ascii*

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int i;
  int j;

  for (i=32; i<128; i++)
    printf("%3d %c\n",i,i);

  return 0;
}
```



## Corrigé de l'exercice n°7.5 — *Un beau tableau*

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  int i;
  int j;

  for (i=4; i<16; i++) {
    for(j=0; j<8; j++) {
        printf("%3d %c  ",i*8+j,i*8+j);
    }
    printf("\n");
  }

  return 0;
}
```

## 7.7 À retenir

Retenez l'équivalence entre les deux programmes suivants :

```
for (i=1; i<100; i=i+2) {
 printf("i vaut:%d",i);
}
```

```
i=1;

while (i<100) {
 printf("i vaut:%d",i);

 i=i+2;
}
```

Chapitre

# 8

# Pointeurs et fonctions

## 8.1 Objectifs

Dans ce chapitre, nous manipulerons deux notions essentielles du langage C : les pointeurs et les fonctions...

## 8.2 Binaire, octets...

### 8.2.1 Binaire

L'ordinateur ne « comprend » que des `0` et des `1`. On dit qu'il travaille en base 2 ou binaire (tout simplement parce que ce système de numération est particulièrement adapté à la technologie électronique utilisée pour construire les ordinateurs). Au contraire des machines, les humains préfèrent généralement la base 10, c'est à dire le système décimal.

La table 8.1 donne l'écriture binaire et décimale de quelques nombres.

Nous voyons dans cette table que le nombre 7 est donc représenté par trois symboles « `1` » qui se suivent : `111`.

| Binaire | Décimal |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 10 | 2 |
| 11 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |
| ... | ... |
| 11111111 | 255 |

TABLE 8.1 - Équivalences entre l'écriture binaire et l'écriture décimale



Chacun de ces « 1 » et « 0 » s'appelle un bit (contraction de l'anglais *binary digit* : chiffre binaire). Par exemple, le nombre « 1010001 » est composé de 7 bits.



Un ensemble de huit bits consécutifs (chacun pouvant prendre la valeur 0 ou 1), s'appelle un octet (*byte* en anglais).



- 11110000 représente un octet
- 10101010 représente un octet
- 100011101 ne représente pas un octet

### 8.2.2 Compter en base 2, 3, 10 ou 16

Vous avez l'habitude de compter en base 10 (décimal). Pour cela, vous utilisez dix chiffres de 0 à 9.

Pour compter en base 2 (binaire), on utilisera deux chiffres : 0 et 1.

Pour compter en base 3, on utilisera 3 chiffres : 0, 1 et 2.

La base 16 (base hexadécimale) est très utilisée en informatique et notamment en langage assembleur. Mais compter en hexadécimal nécessite d'avoir 16 chiffres ! La solution est de compléter les chiffres habituels de la base 10 par des lettres de l'alphabet comme le montre le tableau 8.2.

| Base 16 | Base 10 |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| A | 10 |
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |

TABLE 8.2 - Base 16 et base 10

Dans la pratique, afin de ne pas faire de confusion entre le nombre 10 (en décimal) et ce même nombre en hexadécimal, on précède ce dernier par le préfixe « `0x` » :

```
int i1=10;
int i2=0x10;

printf("i1=%d i2=%d",i1,i2);
```

Ce programme affichera `i1=10 i2=16`.

# 8.3 Variables : pointeurs et valeurs

## 8.3.1 Variables et mémoire

Une variable est une zone mémoire disponible pour y ranger des informations. Par exemple, dans le cas d'une variable `car` déclarée par `char car;`, la zone mémoire disponible sera de 1 octet (exactement une case mémoire). En revanche, une variable de type `int` utilisera au moins 4 octets (la valeur exacte dépend du compilateur), c'est à dire 4 cases mémoires.

À présent, représentons-nous la mémoire comme une unique et longue « rue » remplie de maisons. Chaque case mémoire est une maison. Chaque maison porte un numéro, c'est ce qui permet de définir son adresse postale. Cette adresse est unique pour chaque maison dans la rue. De manière analogue, pour une case mémoire, on parlera d'*adresse mémoire*.

 Par exemple, ce petit programme va afficher l'adresse mémoire d'une variable de type caractère.

```
#include <stdio.h>

int main () {
  char c='A';
  printf ("c contient %c et est stocké à %p\n",c,&c);
  return 0;
}
```

L'exécution nous donne, par exemple :

```
c contient A et est stocké à 0xbf8288a3
```

 Le format « `%p` » permet d'afficher une adresse en hexadécimal.

 Vous pouvez essayer le programme précédent, mais vous n'aurez probablement pas la même adresse mémoire que dans l'exemple.

### 8.3.2 Pointeurs

Pour manipuler les adresses mémoire des variables, on utilise des variables d'un type spécial : le type « pointeur ». Une variable de type pointeur est déclarée ainsi :

```
type* variable;
```

L'espace après `*` peut être omis ou placé avant, peu importe.

C'est le signe `*` qui indique que nous avons affaire à un pointeur. L'indication qui précède `*` renseigne le compilateur sur le type de case pointée, et comment se comporteront certaines opérations arithmétiques [1]. Si `v` est un pointeur vers un `int`, et que nous désirons modifier l'entier pointé, nous écrirons :

```
*v = valeur;
```

`*v` peut être interprété comme « contenu de l'adresse mémoire pointée par `v` ».

 Lisez le programme suivant ainsi que les explications associées.

```
#include <stdio.h>

int main () {
  char car='C';
  char * ptr_car; /* Variable de type pointeur */

  printf("Avant, le caractère est : %c\n",car);
  ptr_car = &car; /* ptr_car = adresse de car */
```

1. Ajouter 1 à un pointeur sur un `int` revient à ajouter 4 à l'adresse, alors qu'ajouter 1 à un pointeur sur un `char` revient à n'ajouter que 1 à l'adresse.

```
  *ptr_car = 'E'; /* on modifie le contenu de l'adresse mémoire */
  printf("Après, le caractère est : %c\n\n",car);

  printf("Cette modification est due à :\n");
  printf("Adresse de car : %p\n",&car);
  printf("Valeur de ptr_car : %p\n",ptr_car);

  return 0;
}
```



Voici ce que cela donne avec notre rue bordée de maisons :

| | |
|---|---|
| `ptr_car = &car;` | `ptr_car` contient l'adresse postale de Monsieur `car` |
| `*ptr_car = 'E';` | On entre chez Monsieur `car` et on y dépose le caractère `E` |
| `printf ("%c",car);` | On va regarder ce qu'il y a chez Monsieur `car` pour l'afficher à l'écran |

Table 8.3 - – Pointeurs et valeurs

On supposera que la variable `car` est stockée à l'adresse `0x0100` ; la variable `ptr_car` est stockée à l'adresse `0x0110`.

Reprenons les deux premières lignes d'exécution du programme :

```
char car='C';
char* ptr_car;
```

La mémoire ressemble alors à ceci :

| Nom de la variable | car | | ptr_car |
|---|---|---|---|
| Contenu mémoire | C | | ? |
| Adresse mémoire (0x) | 100 | | 110 |

Table 8.4 - Stockage de variables (a)

Nous mettons « ? » comme contenu de `ptr_car` car cette variable n'a pas encore été initialisée. Sa valeur est donc indéterminée.

Le programme se poursuit par :

```
printf("Avant, le caractère est : %c\n",car);
ptr_car = &car; /* ptr_car = adresse de car */
```

Il y a donc affichage à l'écran de `Avant, le caractère est : C` puis la mémoire sera modifiée comme ceci :

| Nom de la variable | `car` | | `ptr_car` |
|---|---|---|---|
| Contenu mémoire | C | | 100 |
| Adresse mémoire (0x) | 100 | | 110 |

TABLE 8.5 - Stockage de variables (b)

`ptr_car` n'est donc qu'une variable, elle contient la valeur `100` (l'adresse de la variable `car`).

Finalement, la ligne :

```
*ptr_car = 'E'; /* on modifie le contenu de l'adresse mémoire */
```

conduit à modifier la mémoire comme suit :

| Nom de la variable | `car` | | `ptr_car` |
|---|---|---|---|
| Contenu mémoire | E | | 100 |
| Adresse mémoire (0x) | 100 | | 110 |

TABLE 8.6 - Stockage de variables (c)

Prenez le temps de comprendre tout ce qui précède avant de passer à la suite.

### 8.3.3 Exercice d'application



**Exercice n°8.1** — *Intelligent*

Réalisez un programme équivalent qui change une valeur numérique (`int`) de 10 à 35.



Notons que, dans la pratique, lorsque l'on déclare un pointeur, il est plus prudent de l'initialiser :

```
#include <stdio.h>

int main () {
  char car='C';
  char * ptr_car=NULL;

  printf("Avant, le caractère est : %c\n",car);
  ptr_car = &car; /* ptr_car = adresse de car */
  *ptr_car = 'E'; /* on modifie le contenu de l'adresse mémoire */
  printf("\nAprès le caractère est : %c\n",car);/*on a modifié car*/

  return 0;
}
```

La constante NULL vaut 0. Or un programme n'a pas le droit d'accéder à l'adresse 0. Dans la pratique, écrire prt_car=NULL signifie que l'adresse mémoire pointée est pour l'instant invalide.

On peut se demander à quoi servent les pointeurs ? ! En effet, ceci a l'air compliqué alors que l'on pourrait faire bien plus simple.

Ainsi tout le travail effectué par le programme précédent se résume au code suivant :

```
#include <stdio.h>
int main () {
 char car='C';

 printf("Avant, le caractère est : %c\n",car);

 car='E';
 printf("\nAprès le caractère est : %c\n",car);/*on a modifié car*/

 return 0;
}
```

Nous verrons par la suite que dans certains cas, on ne peut pas se passer des pointeurs, notamment pour certaines manipulations au sein des fonctions...

## 8.4 Fonctions

Une fonction est un petit bloc de programme qui à l'image d'une industrie va créer, faire ou modifier quelque chose. Un bloc de programme est mis sous la forme d'une fonction si celui-ci est utilisé plusieurs fois dans un programme ou simplement pour une question de clarté. De la même manière que nous avions défini la fonction main, une fonction se définit de la façon suivante :

```
<type de sortie> <nom de la fonction> (<paramètres d'appels>) {
   Déclaration des variables internes à la fonction

   Corps de la fonction

   Retour
}
```

Comme indiqué précédemment, il ne faut pas mettre les < et >, qui ne sont là que pour faciliter la lecture.

Voici un exemple de fonction qui renvoie le maximum de deux nombres :

```
int maximum (int valeur1, int valeur2) {
 int max;
 if (valeur1<valeur2)
  max=valeur2;
 else
  max=valeur1;
 return max;
}
```



Le programme complet pourrait être le suivant :

```
01. #include <stdio.h>
02.
03. int maximum (int valeur1, int valeur2) {
04.  int max;
05.  if (valeur1<valeur2)
06.     max=valeur2;
07.  else
08.     max=valeur1;
09.  return max;
10. }
11.
12. int main () {
13.  int i1,i2;
14.  printf("entrez 1 valeur:");
15.  scanf("%d",&i1);
16.  printf("entrez 1 valeur:");
17.  scanf("%d",&i2);
18.  printf("max des 2 valeurs :%d\n",maximum(i1,i2));
19.  return 0;
20. }
```



- Dans la pratique (et en résumant un peu), quand vous tapez `./programme` pour lancer votre programme, l'ordinateur exécute la fonction `main` qui se trouve à la ligne 12. L'ordinateur demande donc ensuite d'entrer deux valeurs qui seront stockées dans `i1` et `i2`.
- Supposons que l'utilisateur ait entré les valeurs `111` et `222`.
- Arrivé à l'exécution de la ligne 18, la machine doit appeler la fonction `printf`, et pour cela, chacun des paramètres de la fonction doit être évalué.
- L'exécution passe alors à la fonction `maximum`, ligne 3. À ce niveau, on peut comprendre intuitivement que la variable `valeur1` prend la valeur de `i1` (c'est-à-dire `111`) et la valeur `valeur2` prend la valeur de `i2` (c'est-à-dire `222`), du fait de l'appel de `maximum(i1,i2)`.
- La fonction `maximum` se déroule. Après avoir passé les lignes `4` à `8`, la valeur de `max` sera de `222`.
- À la ligne 9 on sort donc de la fonction `maximum` pour revenir à l'appel fait par `printf` de la ligne 18. Tous les paramètres de la fonction ayant été évalués (`maximum(i1,i2)` a été évalué à 222), `printf` est exécuté et dans le format, « `%d` » est remplacé par 222.
- Le programme se termine.

### 8.4.1 Type void

Le mot *void* signifie vide. Le type `void` est notamment utilisé comme type de sortie pour les fonctions qui ne retournent aucun résultat (qu'on appelle aussi procédures).

Exemple :

```
/* Fonction affichant un caractère */
void affiche_car (char car) {
 printf ("%c",car);
}
```



Exemple de programme complet :

```
#include <stdio.h>

/* Fonction affichant un caractère */
void affiche_car (char car) {
 printf ("%c",car);
}

int main () {
 char c;
 int i;

 printf("Veuillez entrer un caractère:");
 scanf("%c",&c);

 for (i=0; i<100; i++)
  affiche_car(c);
 return 0;
}
```

### 8.4.2 Variables globales et locales



Les variables déclarées dans les fonctions sont dites locales. Il existe aussi les variables dites globales qui sont déclarées en-dehors de toute fonction (y compris le `main()`).

Les variables globales sont modifiables et accessibles par toutes les fonctions sans avoir besoin de les passer en paramètres. Il est de ce fait extrêmement dangereux d'utiliser des variables globales.

Les variables locales ne sont modifiables et accessibles que dans la fonction où elles sont déclarées. Pour les modifier ou les utiliser dans une autre fonction, il est nécessaire de les passer en paramètres.

#### 8.4.2.1 Variables locales

```
#include <stdio.h>

int carre (int val) {
 int v = 0; /* Variable locale */

 v = val * val;
 return v;
}

int main () {
 int val_retour = 0; /* Variable locale */

 val_retour = carre (2);
 printf("Le carré de 2 est : %d\n", val_retour);

 return 0;
}
```

La variable `val_retour` de main et la variable `v` de `carre` sont toutes deux des variables locales. Le passage des valeurs se fait par copie. Lors du `return v`, on peut imaginer que

c'est le contenu de v qui est renvoyé, puis stocké dans val_retour. La variable v est détruite lorsqu'on revient dans main.

#### 8.4.2.2 Variables globales

```
#include <stdio.h>

int val = 0;
int val_retour = 0;

void carre () {
 val_retour = val * val;
}

int main () {
 val = 2;

 carre ();
 printf("Le carré de 2 est %d\n", val_retour);

 return 0;
}
```

Les variables val et val_retour sont des variables globales. On constate que le programme devient rapidement illisible et difficile à vérifier…

Le conseil à retenir est de ne pas utiliser de variable(s) globale(s) lorsqu'il est possible de s'en passer.

### 8.4.3 Utilisation et modification de données dans les fonctions

L'appel d'une fonction peut s'effectuer à l'aide de paramètres.

Ces paramètres figurent dans les parenthèses de la ligne de titre de la fonction.



Par exemple :

```
#include <stdio.h>

void affiche_Nombre (int no) {
 no=no+1;
 printf ("Le nombre no est : %d\n",no);
}

int main (){
 int a=12;
 affiche_Nombre (a);
 printf("Le nombre a est : %d\n",a);
 return 0;
}
```

Dans le cas qui précède, lors de l'appel de fonction, c'est le *contenu* de a (c'est à dire 12) qui est envoyé dans la fonction, et qui se retrouve donc stocké dans no. Le nombre 12 existe donc en deux exemplaires : une fois dans la variable a de main et une fois dans la variable no de affiche_Nombre. Puis on ajoute 1 dans no (mais pas dans a), et on l'affiche. On voit donc apparaître :

```
Le nombre no est : 13
```

La procédure se termine alors et la variable no est détruite. On revient dans la fonction main qui affiche le contenu de a qui vaut encore 12... On voit donc apparaître :

```
Le nombre a est : 12
```

Il faut donc retenir que les paramètres d'une fonction sont passés *par valeur* et que modifier ces paramètres ne modifie que *des copies* des variables d'origine.

Pour pouvoir modifier les variables d'origine, il ne faut plus passer les paramètres par valeur, mais par adresse, en utilisant les pointeurs, comme c'est le cas dans l'exemple qui suit.



```
#include <stdio.h>

void avance_Position (int* pointeur_int) {
  *pointeur_int = *pointeur_int + 2;
}

int main () {
  int x=0;

  printf("Position de départ : %d\n",x);

  avance_Position (&x);

  printf("Nouvelle position : %d\n",x);

  return 0;
}
```

Imaginons que pointeur_int se trouve en mémoire à l'adresse 20 et x à l'adresse 10 :

| Nom de la variable | x | | pointeur_int |
|---|---|---|---|
| Contenu mémoire | | | |
| Adresse mémoire (0x) | 10 | | 20 |

TABLE 8.7 - Stockage de variables (d)

À présent, déroulons le programme principal :

1. int x=0; : déclaration et initialisation de x à 0
2. Avance_Position (&x); : appel de la fonction Avance_Position (&x); avec la valeur 10 (l'adresse de x) en paramètre
3. void Avance_Position (int* pointeur_int) : on lance cette fonction et pointeur_int vaut donc 10
4. * pointeur_int = (* pointeur_int) + 2; : (*pointeur_int) pointe sur la variable x. L'ancienne valeur de x va être écrasée par sa nouvelle valeur : 0+2

| Nom de la variable | x | | pointeur_int |
|---|---|---|---|
| Contenu mémoire | 0 | | 10 |
| Adresse mémoire (0x) | 10 | | 20 |

TABLE 8.8 - Stockage de variables (e)

5. `printf("Nouvelle position :%d",x);` : on se retrouve avec 2 comme valeur.

Nous devons utiliser des pointeurs pour pouvoir modifier certaines variables *en dehors de l'endroit où elles sont déclarées*. On dit généralement qu'une fonction n'est pas capable de modifier ses arguments. L'usage des pointeurs devient dans ce cas nécessaire... Ainsi, le programme suivant affichera 2 et non 4.

```
#include <stdio.h>

void calcule_double (int x){
  x=x+x ;
}

int main () {
  int i=2;
  calcule_double(i);
  printf("i vaut à présent :%d",i); /* il vaut toujours 2 !!! */

  return 0;
}
```

Pour rendre le programme fonctionnel en utilisant les pointeurs, il faudrait écrire :

```
#include <stdio.h>

void calcule_double (int * p_i){
  *p_i=*p_i+*p_i;
}

int main () {
  int i=2;
  calcule_double(&i);
  printf("i vaut à présent :%d",i);
  return 0;
}
```

Néanmoins, la bonne pratique (celle qui donne les programmes les plus simples à comprendre et les plus faciles à déboguer) serait d'écrire :

```
#include <stdio.h>

int calcule_double (int a){
  a=a+a;
  return a;
}

int main () {
  int i=2;
  i=calcule_double(i);
  printf("i vaut à présent :%d",i);
```

```
  return 0;
}
```



Observez attentivement les trois exemples qui précèdent et soyez sûr d'avoir bien compris pourquoi le premier affiche 2 alors que les deux suivants affichent 4.

**Exercice n°8.2** — *Encore un*



Modifiez le programme précédent afin d'avoir à disposition une fonction qui prend en paramètre un pointeur vers un entier et **modifie** l'entier pointé en lui ajoutant 1.



Ce type d'opération sur les pointeurs est dangereux :

```
int* x;
*x++;
```

augmentera l'adresse de x (on va chez le voisin) et non sa valeur. Pour cela il faut écrire :

```
(*x)++;
```

Ceci est une faute courante, prenez garde …

**Exercice n°8.3** — *Mon beau sapin*



Reprenez le programme du sapin de Noël du chapitre précédent. Écrivez deux fonctions :

- `ramure (int clignes)` qui dessine la ramure du sapin sur clignes de hauteur,
- `tronc (int pos_t)` qui dessine le tronc en position `pos_t` (`pos_t` blancs avant le tronc).

Puis utilisez ces deux fonctions pour dessiner le sapin sur `n` lignes.

### 8.4.4 Prototypes des fonctions

Dans la pratique, lorsqu'on souhaite écrire un programme qui contient de nombreuses fonctions, on essaie de présenter le code de manière propre et structurée. Ainsi, plutôt que cette version :

```
#include <stdio.h>

int max (int x, int y) {
  if (x<y)
    return y;
  else
    return x;
}

int min (int x, int y) {
  if (x<y)
    return x;
  else
    return y;
}
```

```
int main () {
  int i1,i2;
  i1=123;
  i2=1267;
  printf("max : %d\n",max(i1,i2));
  printf("min : %d\n",min(i1,i2));
  return 0;
}
```

on lui préfèrera la version suivante où les prototypes des fonctions disponibles figurent en début de code (un peu comme une table des matières) :

```
#include <stdio.h>

/* PROTOTYPES : */
int max (int x, int y);
int min (int x, int y);

int max (int x, int y) {
  if (x<y)
    return y;
  else
    return x;
}

int min (int x, int y) {
  if (x<y)
    return x;
  else
    return y;
}

int main () {
  int i1,i2;
  i1=123;
  i2=1267;
  printf("max : %d\n",max(i1,i2));
  printf("min : %d\n",min(i1,i2));
  return 0;
}
```

L'intérêt de faire figurer les prototypes en début de programme, en plus d'augmenter la clarté du code, sera vu par la suite.

## 8.5 Corrigés des exercices du chapitre



**Corrigé de l'exercice n°8.1** — *Intelligent*

```
#include <stdio.h>

int main () {
  int val = 10;
  int * ptr_val;

  printf ("Avant le nombre est : %d\n",val);
  ptr_val = &val;
  *ptr_val = 35;
```

```
  printf ("Après le nombre est : %d\n",val);

  return 0;
}
```



### Corrigé de l'exercice n°8.2 — *Encore un*

```
#include <stdio.h>

void ajoute_un (int* pointeur_int) {
  *pointeur_int = *pointeur_int + 1;
}

int main () {
  int i=10;

  printf ("i=%d\n",i);
  ajoute_un (&i);
  printf ("i=%d\n",i);

  return 0;
}
```



### Corrigé de l'exercice n°8.3 — *Mon beau sapin*

```
#include <stdio.h>
#include <stdlib.h>

// Dessin de la ramure du sapin
void ramure (int clignes) {
  int i=0, j=0;

  for (i=1; i<=clignes; i++) {
    for (j=0; j<clignes-i; j++)
      printf("*");
    for (j=1; j<= (i*2-1); j++)
      printf(" ");
    printf("\n");
  }
}

// Dessin du tronc du sapin
void tronc (int pos_t) {
  int i=0, j=0;
  for (j=1; j<=3; j++) {
    for (i=1; i<=pos_t; i++)
      printf (" ");

      printf ("@@@\n");
  }
}

int main () {
  int nb_lig = 15;

  ramure (nb_lig);
  tronc (nb_lig - 2);

  return 0;
}
```

# 8.6 À retenir

## 8.6.1 Les différentes bases de numération

Il est souhaitable de retenir que :

- compter en base 2 (binaire) revient à n'utiliser que des `0` et des `1` ;
- un octet est formé de 8 bits : `11110101` ;
- la base 10 est celle qu'on emploie tous les jours ;
- en base 16 (base hexadécimale), des lettres sont utilisées en plus des 10 chiffres (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

## 8.6.2 Pointeur

Un pointeur est une variable faite pour contenir une adresse mémoire, souvent l'adresse d'une autre variable.

## 8.6.3 Structure d'un programme C

Pour finir, voici deux programmes qui reprennent l'essentiel de ce qui a été vu :

```
#include <stdio.h>

int main () {
  int i=100;
  int * pointeur_sur_i=NULL;

  pointeur_sur_i=&i;

  *pointeur_sur_i=200;
  printf ("i vaut à présent %d\n",i);

  return 0;
}
```

Le programme précédent revient à stocker `200` dans la variable `i`.

```
#include <stdio.h>

/* Prototypage */
void avance_Position (int* x);

void avance_Position (int* x) {
  (*x)+=2;
}

int main () {
  int i=0;
  int x=0;

  printf("Position de départ : %d\n",x);

  for (i = 0; i<5; i++) {
```

```
    avance_Position (&x);
    printf ("Nouvelle position : %d\n",x);
  }

  return 0;
}
```

# Chapitre 9

# Tableaux et chaînes de caractères

## 9.1 Objectifs

En C, une chaîne de caractères est équivalente à un tableau de caractères. Ce chapitre introduit ces deux notions (chaînes et tableaux) tout en vous faisant approcher de la gestion de la mémoire.

## 9.2 Tableaux

### 9.2.1 Définition

Un tableau est un ensemble d'éléments rangés en mémoire dans des cases consécutives (voir Table 9.1). Un tableau peut être constitué de plusieurs lignes et colonnes. Nous n'utiliserons dans un premier temps que les tableaux à une seule ligne.



Notez que les cases d'un tableau sont numérotées à partir de `0` en langage C.

| Numéro de case | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Contenu | A | B | C | D | E | F | G | H |

Table 9.1 - Tableau de caractères

### 9.2.2 Déclaration

Un tableau se déclare de la manière suivante :

```
<type> <nom du tableau> [<taille du tableau>];
```



```
/* Déclaration d'un tableau de 10 caractères */
char tab_char [10];

/* Déclaration d'un tableau de 10 entiers */
int tab_int [10];
```

### 9.2.3 Utilisation

On accède à une case du tableau en mettant le nom du tableau, suivi d'un crochet ouvrant « [ » puis un numéro de case et un crochet fermant : « ] ». Cela donne, par exemple :



```
/* déclarations */
int tab_int[10]; /* tableau de 10 cases (0 à 9) d'entiers */
char tab_char[10]; /* tableau de 10 cases (0 à 9) de caractères */

/* utilisation */
tab_char[3]='C'; /* Initialisation de la case 3 (la quatrième) de tab_char →
    ↪ */
tab_int[6]=10; /* Initialisation de la case 6 (la septième) de tab_int */
tab_int[7]=tab_int[6] * 2; /* La case 7 (la huitième) contiendra donc 20  →
    ↪ (10*2) */
```



N'oubliez pas que le compilateur ne vérifie pas que vous utilisez le tableau dans ses limites. Il vous est donc possible d'écrire à l'extérieur de votre tableau, donc chez le voisin. C'est l'un des bugs les plus courants de la programmation en C.

## 9.3 Chaînes de caractères

Les chaînes de caractères sont des tableaux de caractères suivis du `0` (zéro ; ne pas confondre avec le caractère O de « Oh la la » par exemple...) qui est considéré lui aussi comme un caractère. Une chaîne s'écrit donc : contenu utile de la chaîne + valeur `0`.



« Eric » s'écrit dans un tableau de 5 caractères de la façon suivante (l'usage du `\0` sera expliqué par la suite) :

| Caractère | 'E' | 'r' | 'i' | 'c' | '\0' |
|---|---|---|---|---|---|
| Code ASCII | 69 | 114 | 105 | 99 | 0 |
| Case | 0 | 1 | 2 | 3 | 4 |

TABLE 9.2 - – Chaînes de caractères

### 9.3.1 Déclaration d'une chaîne de caractères

Une chaîne de caractères se déclare sous la forme d'un tableau de caractères de longueur fixe. Attention, comme signalé auparavant, si vous dépassez la longueur de tableau, vous écrivez chez le voisin.

Ainsi :

```
char m_chaine [20];
```

permettra d'enregistrer des chaînes de 19 caractères maximum (`20-1` pour le `0` de fin de chaîne).

Il est possible de déclarer une chaîne de caractères sans en spécifier la longueur de départ de la façon suivante :

```
char chaine [] = "Eric";
```

De cette façon, la chaîne fera exactement la longueur nécessaire pour stocker « Eric » et le `0` final soit 4+1=5 octets.

### 9.3.2 Affichage d'une chaîne de caractères

Une chaîne de caractères s'affiche grâce à la commande `printf` et le format `%s`.

Ainsi :

```
printf("%s",chaine);
```

affichera le contenu de `chaine`.

### 9.3.3 Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères s'obtient par la fonction `strlen` (disponible au travers de la bibliothèque `string`). Le `0` de fin de chaîne n'est pas compté dans cette longueur.



```
#include <stdio.h>
#include <string.h>

int main () {
  char ch [] = "toto" ;
  printf("La longueur de %s est : %d",ch,strlen(ch));
  return 0;
}
```

Affichera : « `La longueur de toto est : 4` » à l'écran.

### 9.3.4 Initialisation d'une chaîne de caractères



Le format `%p`, que nous avons déjà vu, permet l'affichage d'un `void *` (pointeur sur un type void) et va nous servir par la suite à afficher les adresses mémoires…

Le programme suivant :

```
char tab[10];
printf("adresse où commence tab=%p",&tab[0]);
```

affichera la même chose que ce programme :

```
char tab[10];
printf("adresse où commence tab=%p",tab);
```

On voit donc que `tab` et `&tab[0]` sont égaux. En revanche, le programme suivant est incorrect :

```
char tab[10];
tab="coucou";
```

En effet, `tab` désigne l'adresse où débute le tableau en mémoire. Dans l'affectation `tab="coucou";` le membre de gauche désigne une adresse alors que le membre de droite désigne une chaîne de caractères ; les deux n'étant pas du même type, le compilateur C le refuse…

Pour initialiser une chaîne de caractères, il est possible d'utiliser la fonction `strcpy`. Cette fonction nous impose une nouvelle fois d'ajouter le fichier d'en-tête `string.h` :

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char line[80];
  strcpy(line,"un exemple de chaine initialisée...");
  printf ("%s\n",line);
  return 0;
}
```

Une recopie de la chaîne « `un exemple de chaine initialisée…` » caractère par caractère est effectuée en démarrant à l'adresse où `line` se trouve stockée en mémoire (le `'\0'` final est copié lui aussi).

Enfin, si l'on souhaite lire une chaîne directement au clavier, on peut utiliser la fonction `scanf` :

```
#include <stdio.h>

int main(void) {
  char line[80];
  printf("veuillez entrer votre chaine:");
  scanf("%s",line);

  /* scanf("%s",&line[0]) ferait la même chose */

  printf("la chaine saisie vaut :%s",line);
  return 0;
}
```



Notons que les chaînes de caractères saisies de cette manière ne peuvent comporter ni espaces, ni tabulations.

### 9.3.5 Exercices

**Exercice n°9.1** — *Affichez une chaine de caractères*



Lisez l'intégralité de l'exercice avant de démarrer...

- En utilisant une boucle `for`, remplissez un tableau de 10 caractères avec les lettres de l'alphabet en commençant par A (code ASCII 65) ; le tableau devra donc contenir ceci :

| Case | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Contenu | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 0 |

TABLE 9.3 - – Remplissez un tableau...

- Faites afficher la chaîne de caractères ainsi obtenue
- Faites afficher chaque caractère du tableau sous la forme « Caractère n°`0` : A ».



Il est possible d'écrire `tab_car [i] = code_ascii;` où `code_ascii` est un entier représentant le code ASCII du caractère désigné.



Pour faire afficher un seul caractère, on utilisera la syntaxe suivante :

```
int pos=0; /* Position dans le tableau */
printf ("Caractère numéro %d : %c",pos, tab_car [pos]);
```



La valeur `0` peut être assignée à un élément de tableau de la façon suivante :

```
tab_car [la bonne position] = 0;
```

ou encore (étant donné que le caractère `'\0'` désigne la même chose que `0`) par :

```
tab_car [la bonne position] = '\0';
```

On préfèrera généralement cette dernière solution qui est plus explicite et montre bien que l'on travaille avec des chaînes de caractères.

Enfin, voici un exemple de programme qui permet d'initialiser élégamment une chaîne de caractères à « ABCDEFGHIJKLMNOPQRSTUVWXYZ » puis de l'afficher à l'écran :



```
#include <stdio.h>
int main () {
  int i=0;
  int pos_tab=0;
  char tab_alpha[27];
  for (i='A';i<='Z';i++){
    tab_alpha[pos_tab]=i;
    pos_tab++;
  }
  tab_alpha[26]=0;
  printf("%s\n",tab_alpha);
  return 0;
}
```

### 9.3.6 La fonction gets : saisie d'une chaîne de caractères

La fonction `gets` permet de saisir une chaîne de caractères validée par la touche ENTREE . Attention, la touche ENTREE elle-même n'est pas enregistrée dans le tableau de caractères.



Lorsque vous compilez votre programme par :

```
gcc -o essai essai.c
```

vous verrez peut être un avertissement du type :

```
warning : gets may be dangerous
```

Si c'est le cas, ne vous inquiétez pas. Le compilateur craint simplement qu'à l'exécution, l'utilisateur saisisse une chaîne de caractères qui soit plus longue que l'espace qui a été réservé pour la stocker (80 caractères dans l'exemple ci-dessous), auquel cas des problèmes risquent d'apparaître. L'utilisation de `gets` est donc à proscrire dans un cadre « professionnel ».



```
#include <stdio.h>

int main(void) {
  char line[81];
  /* 81 : taille arbitraire supposée suffisante
     Une ligne écran = 80 caractères + 1 case
     pour le '\0' de fin de chaîne */

  printf( "Saisissez une chaîne de caractère :\n" );
  gets( line );
  /* La frappe de l'utilisateur sera enregistrée dans
     line, on suppose qu'il ne frappera pas plus de
     80 caractères, sinon aïe aïe aïe */
```

```
  printf( "\nLa chaîne saisie est : \n%s\n", line );
  return 0;
}
```

Voici un exemple d'exécution :

```
Saisissez une chaîne de caractères :
Bonjour !

La chaîne saisie est :
Bonjour !
```

Notons qu'il n'y a qu'un seul passage à la ligne (celui affiché par la fonction `printf`).

### 9.3.7 Passage d'une chaîne de caractères en paramètres

Pour passer un tableau (et donc une chaîne de caractères) en paramètre à une fonction, nous devons simplement donner *l'adresse du début du tableau*. Les deux fonctions suivantes sont donc équivalentes :

```
int ma_saisie (char chaine[]) {
  /* ...Faire ce qu'il faut... */
  return 0;
}

int main () {
  char ma_chaine [30];
  ma_saisie (ma_chaine);
  return 0;
}
```

```
int ma_saisie (char* chaine) {
  /* ...Faire ce qu'il faut... */
  return 0;
}

int main () {
  char ma_chaine [30];
  ma_saisie (ma_chaine);
return 0;
}
```

En fait, dans la pratique, l'écriture suivante :

```
int ma_saisie (char chaine[]) {
```

est équivalente à celle-ci :

```
int ma_saisie (char * chaine) {
```

Nous reviendrons longuement, par la suite, sur cette quasi-équivalence pointeurs/tableaux... ne vous inquiétez pas si vous ne comprenez pas tout pour l'instant...

## 9.4 Quelques fonctions utiles

### 9.4.1 La fonction strcat

La fonction strcat (<s>, <t>) ajoute la chaîne de caractères <t> à la fin de <s> (on appelle cela une concaténation).

Le programme suivant affichera la chaîne « Bonjour Paul » à l'écran :

```
#include <stdio.h>
#include <string.h>

int main () {
  char chaine1[20]="Bonjour ";
  char chaine2[20]="Paul";

  strcat(chaine1,chaine2); /* ajoute chaine2 à la fin de chaine1 */

  printf("%s\n",chaine1);

  return 0;
}
```

On remarquera qu'il est important de dimensionner chaine1 à une taille suffisante, sans quoi on pourrait avoir des difficultés pour stocker la chaîne « Bonjour Paul » dans chaine1.

### 9.4.2 La fonction strncpy

La fonction strncpy (<s>, <t>, <n>) est presque similaire à strcpy mais copie au plus <n> caractères de la chaîne <t> au début de <s>.

```
#include <stdio.h>
#include <string.h>

int main () {
  char chaine1[20]="Bonjour ";
  char chaine2[20]="Edouard";

  strncpy(chaine1,chaine2,2); /* recopie 2 caractères de chaine2 à l'adresse de →
     ↪ chaine1 */

  printf("%s\n",chaine1);

  return 0;
}
```

...affichera :

```
Ednjour
```

### 9.4.3 La fonction strncat

La fonction `strncat (<s>, <t>, <n>)` ajoute au plus `<n>` caractères de la chaîne `<t>` à la fin de `<s>`.

```
#include <stdio.h>
#include <string.h>

int main () {
  char chaine1[20]="Bonjour ";
  char chaine2[20]="Edouard";

  strncat(chaine1,chaine2,2); /* ajoute les 2 premiers caractères de chaîne2 à la →
     ↪  fin de chaine1 */

  printf("%s\n",chaine1);

  return 0;
}
```

...affichera :

```
Bonjour Ed
```

### 9.4.4 La fonction strcmp

La fonction `strcmp (<s>, <t>)` compare les chaînes de caractères `<s>` et `<t>` de manière lexicographique et fournit un résultat :

- *nul* (`0`) si `<s>` est égale à `<t>`
- *négatif* si `<s>` précède `<t>`. Par exemple, `strcmp("AAAA","BBBB")` renverrait -1
- *positif* si `<s>` suit `<t>`. Par exemple, `strcmp("BBBB","AAAA")` renverrait +1

### 9.4.5 Les fonctions sprintf et sscanf

Nous terminerons par deux fonctions très utiles.

`sprintf(<chaine cible>,<chaine de formatage>,<expr1>,<expr2>,…)`

La fonction `sprintf` renvoie une valeur négative en cas d'erreur et le nombre de caractères stockés dans la chaîne cible sinon.



Dans cet exemple, la fonction va convertir l'entier `i` en chaîne de caractères et la stocker dans la variable `s`. À l'arrivée, `s` contiendra « 15 ».

```
char s[200];
int i=15;
int code;

code=sprintf(s,"%d",i);
```



La fonction `sscanf` fait le contraire.

```
char s[]="12.5 12.3 11.6";
float a,b,c;
int code;

code=sscanf(s,"%f%f%f",&a,&b,&c);
```

Les variables numériques `a`, `b` et `c` contiendront respectivement : `12.5 12.3` et `11.6`.

En cas d'erreur, `sscanf` renvoie une valeur négative. S'il n'y a pas eu d'erreur, c'est le nombre de variables affectées qui est renvoyé.

## 9.5 Tableaux à 2 dimensions

Un tableau à 2 dimensions se déclare de la façon suivante :

```
<type> <nom du tableau> [<taille dimension 1>] [<taille dimension 2>];
```



Par exemple :

```
int table[5][5]; /* représente un tableau d'entiers de 5 lignes et 5 →
    ↪ colonnes.*/
```

Ou bien :

```
float tab[3][2]= {{ 1.2, -1.3 },
                  { 8.5, 12.4 },
                  { -123.0, 4.0 }};
```

Voici à présent un programme qui affiche le contenu d'un tableau à deux dimensions :

```
#include <stdio.h>

int main () {
  int tab[5][10];
  int i,j;
  /* Pour chaque ligne ... */
  for (i=0; i<5; i++){
    /* ... considérer chaque case */
    for (j=0; j<10; j++)
      printf("%d ", tab[i][j]);
    /* Retour à la ligne */
    printf("\n");
  }
  return 0;
}
```

Si on souhaite initialiser ce tableau avec des valeurs lues au clavier, voici comment faire :

```
#include <stdio.h>

int main () {
  int tab[5][10];
  int i,j;
  /* Pour chaque ligne ... */
  for (i=0; i<5; i++){
    /* ... considérer chaque case */
    for (j=0; j<10; j++)
      scanf("%d", &tab[i][j]);
    /* Retour à la ligne */
    printf("\n");
  }
  return 0;
}
```

 Nous aurions pu écrire ce programme sous la forme suivante :

```
#include <stdio.h>

#define LIGNES 5
#define COLONNES 10

int main () {
  int tab[LIGNES][COLONNES];
  int i,j;
  /* Pour chaque ligne ... */
  for (i=0; i<LIGNES; i++){
    /* Considérer chaque case */
    for (j=0; j<COLONNES; j++)
      scanf("%d", &tab[i][j]);
    /* Retour à la ligne */
    printf("\n");
  }
  return 0;
}
```

L'usage de `#define LIGNES 5` demande au compilateur de parcourir tout le programme et de faire une sorte de *chercher-remplacer* : *chercher* la chaîne `LIGNES` et *remplacer* par 5. Il fera ensuite pareil pour `COLONNES`.

Il ne faut pas mettre des « ; » à la fin des `#define`.

```
#include <stdio.h>

#define LIGNES 5;
#define COLONNES 10;
```

En effet, les rechercher-remplacer aboutiraient au programme suivant qui ne se compilerait pas :

```
int main () {
  int tab[5;][10;];/* compile pas !!! */
  int i,j;
  /* Pour chaque ligne ... */
  for (i=0; i<5;; i++){/* compile pas !!! */
    /* ... considérer chaque case */
```

```
    for (j=0; j<10;; j++)/* compile pas !!! */
      scanf("%d", tab[i][j]);
    /* Retour à la ligne */
    printf("\n");
  }
  return 0;
}
```

## 9.6 Correction des exercices



**Corrigé de l'exercice n°9.1** — *Affichez une chaine de caractères*

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  /* 10 caractères + 0 de fin de chaîne */
  char tab [11];
  int i=0; /* compteur */

  /* Remplissage du tableau avec les caractères */
  for (i=0; i<10; i++)
    tab[i] = 65 + i; // ou tab[i]='A'+i;

  /* Ajout du 0 de fin de chaine*/
  tab[10] = 0;
  /* Affichage de la chaîne*/
  printf("tab : %s\n",tab);

  /* Saut d'une autre ligne */
  printf("\n");

  /* Affichage de chacun des caractères */
  for (i=0; i<10; i++)
    printf("Caractère numero %d: %c\n",i,tab [i]);

  return 0;
}
```

## 9.7 À retenir

Voici pour finir un petit programme qui reprend l'essentiel de ce qui a été vu. Il permet de lire une chaîne de caractères (`chaine1`) au clavier puis de recopier cette chaîne dans une autre (`chaine2`) et de l'afficher à l'écran :

```
#include <stdio.h>
#include <string.h>

int main () {
  char chaine1[81]; /* 80 caractères + '\0' */
  char chaine2[81];

  printf("Veuillez entrer votre chaine de caractères : ");
  scanf("%s",chaine1); /* identique à scanf("%s",&chaine1[0]); */
```

```
  strcpy(chaine2,chaine1); /* !!! attention à l'ordre !!! */

  printf("chaine2 vaut: %s \n",chaine2);

  strcpy(chaine1,""); /* et pas chaine1=""; !!!!!!! */
  strcat(chaine1,"Pierre ");
  printf("Veuillez entrer votre chaine de caractères");
  scanf("%s",chaine2);
  strcat(chaine1,chaine2);
  strcat(chaine1," Paul Jacques...");
  printf("chaine1 vaut: %s \n",chaine1);

  return 0;
}
```

Chapitre

# 10

# Structures et fichiers

## 10.1 Les types synonymes

Cette nouvelle notion de type synonyme va nous servir d'ici peu. Voyons de quoi il s'agit.



Il est possible grâce au mot-clé `typedef` de définir un synonyme pour un type déjà existant. Ainsi la définition suivante :

```
typedef int entier;
```

définit un nouveau type appelé `entier` ayant les mêmes caractéristiques que le type prédéfini `int`. Une fois cette définition réalisée, nous pouvons utiliser ce nouveau type pour définir des variables et nous pouvons mélanger les variables de ce type avec des variables entières dans des expressions.

```
typedef int entier;
entier e1=23, e2=5, te[7]={1,2,3,4,5,6,7};
int i;
i = e1 + e2;
te[3] = i - 60;
```

## 10.2 Structures

Une structure est un objet composé de plusieurs champs qui sert à représenter un objet réel ou un concept. Par exemple une voiture peut être représentée par les renseignements suivants : la marque, la couleur, l'année, etc.



Nous pouvons définir une structure ainsi :

*Solution 1 :*

```
struct nom_de_la_structure {
 /* Définition de la structure */
} nom_du_type;
```

Ceci fait, le nouveau type de données sera `struct nom_du_type` et nous pourrons déclarer une variable ainsi :

```
struct nom_du_type nom_variable;
```

Cependant, la répétition du mot-clé `struct` est rapidement ennuyeuse. Nous préférerons donc souvent la syntaxe suivante.

*Solution 2 :*

```
typedef struct {
 /* Définition de la structure */
} nom_du_type;
```

Cette fois-ci, le nouveau type de données s'appelle `nom_du_type` (nous avons créé la structure et en même temps nous avons défini un synonyme avec `typedef`).

Nous déclarerons une variable ainsi :

```
nom_du_type nom_variable;
```



En pratique, cela donne :

```
#define LONGUEUR 40

struct personne{
 char nom [LONGUEUR];
 char prenom [LONGUEUR];
 int  age;
};

struct personne p;
```

```
#define LONGUEUR 40

typedef struct {
 char nom [LONGUEUR];
 char prenom [LONGUEUR];
 int  age;
} personne;
```

```
personne p;
```

La seconde solution est plus simple et plus élégante à l'usage.



L'accès aux éléments d'une structure, que nous appelons aussi champ, se fait selon la syntaxe :

`nom_de_variable.nom_du_champ`



Par exemple :

```
#include <stdio.h>

typedef struct {
  char nom [40];
  char prenom [20];
  int  age;
} personne;

int main () {
  personne p;
  printf("Veuillez entrer le nom de la personne:");
  scanf("%s",p.nom);

  printf("Veuillez entrer le prénom de la personne:");
  scanf("%s",p.prenom);

  printf("Veuillez entrer l'âge de la personne:");
  scanf("%d",&p.age); /* ne pas oublier le & !!! */

  printf("Voici les caractéristiques de cette personne:\n");
  printf("nom=%s\n",p.nom);
  printf("prenom=%s\n",p.prenom);
  printf("age=%d\n",p.age);

  return 0;
}
```

## 10.3 Bases sur les fichiers

Tout ce qui est enregistré sur votre disque dur ou presque est un fichier, et porte un nom.

Il est possible de créer, de lire ou d'écrire dans des fichiers. Notez que certains fichiers peuvent être protégés en lecture, en écriture ou les deux.

Voici un programme que nous allons détailler :

```
#include <stdio.h>
#include <stdlib.h>
int main () {
 FILE *p_fichier; /* pointeur sur fichier*/
 char nom_fichier[20], nom_personne[20];
```

```
06.  int i, nbr_enregistrements;
07.  /* 1ère étape : Création et remplissage du fichier */
08.  printf("Quel est le nom du fichier à créer ? ");
09.  scanf("%s", nom_fichier);
10.
11.  /* w: write r: read a: append*/
12.  p_fichier = fopen(nom_fichier, "w");
13.  if (p_fichier == NULL) {
14.   printf("Erreur de création du fichier \n");
15.   exit(-1); // Abandonner le programme
16.  }
17.
18.  printf("Nombre de personnes à stocker ? : ");
19.  scanf("%d", &nbr_enregistrements);
20.
21.  for (i = 0; i<nbr_enregistrements; i++) {
22.   printf("Entrez le nom de la personne : ");
23.   scanf("%s", nom_personne);
24.   fprintf(p_fichier, "%s\n", nom_personne);
25.  }
26.  fclose(p_fichier);
27.
28.  /* 2ème étape : Lecture et affichage du fichier */
29.  p_fichier = fopen(nom_fichier,"r"); /* read */
30.  if (p_fichier == NULL) {
31.   printf("\aErreur d'ouverture sur le fichier \n");
32.   exit(-2); // Abandonner le programme
33.  }
34.
35.  while (!feof(p_fichier)) {
36.   fscanf(p_fichier, "%s ", nom_personne);
37.   printf("Nom : %s\n", nom_personne);
38.  }
39.  fclose(p_fichier);
40.
41.  return 0;
42. }
```

*Explications :*

- Ligne 4 : une variable `p_fichier` est créée ; elle va pointer sur un type `FILE`. Sans entrer dans les détails, le type `FILE` est un type structure (vu au paragraphe précédent) qui permet de décrire un fichier.
- Ligne 9 : l'utilisateur va saisir une chaîne au clavier. Cette dernière sera stockée dans la variable `nom_fichier`. Supposons pour fixer les idées que l'utilisateur tape au clavier `familles.txt`. Le fichier qui sera par la suite créé portera ce nom.
- ligne 12 : `fopen` va créer une sorte de lien entre le fichier du disque dur qui s'intitule `familles.txt` et la variable `p_fichier`. Ainsi dans la suite, vous allez faire des opérations sur la variable `p_fichier` et toutes ces opérations seront répercutées au niveau du fichier `familles.txt`. Dans ce cas précis, les 3 opérations suivantes peuvent être réalisées :
  - `p_fichier=fopen(nom_fichier, "w");` : si le fichier `familles.txt` existe déjà, il est purement et simplement écrasé puis réinitialisé à vide. S'il n'existe pas encore, le fichier est crée, pour l'instant il est vide.
  - `p_fichier=fopen(nom_fichier, "r");` : si le fichier `familles.txt` existe déjà, il est simplement ouvert en lecture (*read*). L'ordinateur se positionne sur le premier caractère du fichier. Si le fichier n'existe pas (typiquement, nous nous sommes trompé de nom), la fonction `fopen` renvoie alors `NULL`.

  - `p_fichier=fopen(nom_fichier, "a");` : si le fichier `familles.txt` existe déjà, il est simplement ouvert. Ensuite, l'ordinateur se positionne sur la fin de ce fichier, prêt à ajouter quelque chose après la dernière ligne. Nous comprenons mieux le `"a"` : append. Si le fichier n'existe pas, il est créé, et il est donc vide.
- Ligne 13 : il est toujours prudent de faire ce test. Le pointeur sera nul s'il y a eu un problème lors de l'accès au fichier (nom incorrect pour l'ouverture en lecture, accès en écriture impossible…)
- Ligne 15 : sortie catastrophe, le programme s'arrête immédiatement. La valeur `-1` est renvoyée au système d'exploitation. Il est à noter que l'usage de la fonction `exit` impose d'ajouter la ligne `#include <stdlib.h>`.
- Ligne 23 : l'ordinateur lit au clavier le nom d'une personne.
- Ligne 24 : en fait, un `fprintf` n'est pas très différent d'un `printf`. La seule différence est qu'au lieu d'être écrite sur l'écran, la chaîne `nom_personne` sera écrite dans le fichier `familles.txt`.
- Ligne 26 : on referme le fichier pour indiquer au programme C que l'on a fini de travailler sur `familles.txt` pour l'instant. Il faut toujours penser à faire cette opération.
- Ligne 29 : ré-ouverture du fichier, en lecture, cette fois-ci. Si le `fopen` se passe bien (ce que nous pouvons supposer !), l'ordinateur se positionne alors au début de la 1re ligne du fichier.
- Ligne 34 : `feof` désigne l'abréviation de file end of file. Donc cette ligne se traduit par : *tant que l'on n'atteint pas la fin du fichier désigné par p_fichier…*

Enfin, voici une autre fonction qui peut se montrer très utile :

```
char *fgets(char *ligne, int maxligne, FILE *p_fichiers)
```

La fonction `fgets` lit à partir du fichier au maximum `maxligne -1` caractères et les stocke dans la chaîne de caractères `ligne`.

La lecture s'arrête sur `\n` qui est alors inclus dans la chaîne. La chaîne est complétée par `\0`. La fonction renvoie `NULL` si la fin de fichier est atteinte.



Voici un exemple de programme qui va simplement afficher le contenu du fichier `essai.txt` à l'écran (lisez-le puis étudiez la remarque qui le suit) :

```
#include <stdio.h>

/* Définition de constante */
#define maxligne 100

char ligne[maxligne];
FILE *p_fichier;
int main() {
  p_fichier=fopen("essai.txt","r");
  while (! feof(p_fichier)) {
    fgets(ligne,maxligne,p_fichier);
    if (! feof(p_fichier))
      printf("J'ai lu :%s\n",ligne);
  }
  fclose(p_fichier);
  return 0;
}
```

Le test suivant peut paraître curieux :

```
if (! feof(p_fichier))
 printf("J'ai lu :%s\n",ligne);
```

En fait, il est nécessaire du fait que la fonction

`feof(p_fichier)`

renverra vrai si l'indicateur de fin de fichier du flux `pfichier` est positionné, c'est-à-dire s'il y a déjà eu une lecture infructueuse (par `fgets` par exemple).

Ainsi, lorsque le `fgets` lit la dernière ligne du fichier, un appel, dans la foulée à la fonction `feof(p_fichier)` renverra faux. Ce n'est que si nous refaisons un `fgets` (qui sera donc infructueux) que là, le test `feof(p_fichier)` renverra vrai. Donc finalement, nous voyons bien le problème : pour la toute dernière ligne, le `fgets` va échouer et l'instruction `printf("J'ai lu :%s\n",lignes)`, si elle était appelée, pourrait bien renvoyer n'importe quoi !

## 10.4 Fichiers et structures



Voici un exemple qui mêle fichiers et structures :

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
 char nom [40];
 char prenom [20];
 int age;
} personne;

int main() {
 FILE *p_fichier; /* pointeur fichier */
 /* Créer et remplir le fichier */

 p_fichier = fopen("essai.txt","w");
 if (p_fichier == NULL) {
  printf("\aImpossible de créer le fichier \n");
  exit(-1); // Abandonner le programme
 }

 personne p;
 printf("Veuillez entrer le nom de la personne:");
 scanf("%s",p.nom);
 printf("Veuillez entrer le prénom de la personne:");
 scanf("%s",p.prenom);

 printf("Veuillez entrer l'âge de la personne:");
 scanf("%d",&p.age); /* ne pas oublier le & !!! */

 fprintf(p_fichier, "%s\n",p.nom);
 fprintf(p_fichier, "%s\n",p.prenom);
 fprintf(p_fichier, "%d\n",p.age);

 fclose(p_fichier);
 return 0;
}
```

Chapitre

# 11

# Débogage dun programme

## 11.1 Objectif

L'objectif de ce chapitre est de vous aider à traquer les bugs et à les corriger ;-)...

## 11.2 Deux types d'erreurs

On fera la différence entre les deux types d'erreurs suivants :

- erreur de compilation ou d'édition de liens
- erreur d'exécution

### 11.2.1 Erreur à la compilation

Une erreur lors de la compilation est provoquée par l'exécution de la commande `gcc -o` :



Vous tapez `gcc -o essai essai.c` pour le programme suivant :

```
#include <stdio.h>
int main () {
 floatt f ;

 return 0 ;
}
```

Le compilateur vous renvoie un message du type :

```
essai.c : 3 : 'floatt' undeclared (first use in this function)
```

Une erreur d'édition de liens survient lorsque, par exemple, on utilise une fonction d'une bibliothèque séparée, sans penser à joindre la bibliothèque lors de la création de l'exécutable.

### 11.2.2 Erreur d'exécution

Une erreur d'exécution est provoquée lors de l'exécution du programme, c'est à dire lorsque la compilation et l'édition de liens se sont bien réalisées mais qu'il y a un problème lorsque l'on teste l'exécutable.



```
#include <stdio.h>
int main () {
 float f=1/0;

 return 0 ;
}
```

La compilation ne pose pas de problèmes. Lors de l'exécution du programme (`./essai`), l'ordinateur affiche un message du type : `Floating point exception`. En effet, une division par zéro est interdite.

## 11.3 Un phénomène surprenant…



Saisissez le programme suivant, compilez-le et enfin testez-le…

```
#include <stdio.h>
int main () {
 printf ("Je suis ici") ;
 while (1)
   ;

 return 0 ;
}
```

Rien ne s'affiche ? ! En fait, pour des raisons d'optimisation système, un `printf()` est conservé dans un buffer de ligne jusqu'à réception d'un `\n` ou exécution d'un `fflush()`.

 Essayez :

```
#include <stdio.h>
int main () {
 printf ("Je suis ici\n") ;
 while (1)
   ;

 return 0 ;
}
```

## 11.4 La chasse aux bugs...

La démarche pour corriger un bug [1] est la suivante.

### 11.4.1 Localiser et produire le bug

Prenons l'exemple du programme suivant qui n'affiche rien pour la raison précédemment évoquée (le problème de l' `\n`) :

```
#include <stdio.h>

int main () {
 int i ;
 for (i=0 ; i<100 ; i++)
   printf("i=%d",i) ;
 while (1)
   ;
 return 0 ;
}
```

Le rajout de « mouchards » (un mouchard est ici simplement un `printf`) dans le programme nous permettra de localiser le bug.

```
#include <stdio.h>
int main () {
 int i ;
 printf("1) Je suis ici\n") ; /* 1 er mouchard */

 for (i=0 ; i<100 ; i++)
   printf("i=%d",i) ;
 printf("2) Je suis ici\n") ; /* 2 eme mouchard */

 while (1)
   ;
 return 0 ;
}
```

1. Un bug est un défaut dans un programme produisant des anomalies de fonctionnement.

### 11.4.2 Corriger le bug

La phase précédente vous a permis de savoir d'où venait le problème, en revanche, celle-ci ne l'a pas corrigé.

Prenons l'exemple suivant :

```
#include <stdio.h>
int main () {
  int i,j;
  i=0;
  j=0;
  printf("1) Je suis ici\n") ;
  if ((i==0) && (i=j)) {
    printf("2) Je suis ici\n");
    ...
  }
  return 0 ;
}
```

Vous êtes persuadé que l'ordinateur devrait afficher `2) Je suis ici` or il n'en est rien ? ! Vous en êtes à invoquer un bug dans le compilateur ? ! L'ordinateur ne fait que ce que vous lui demandez. Vous allez donc affiner votre traçabilité en plaçant des mouchards pour voir précisément le contenu des variables.

```
#include <stdio.h>
int main () {
  int i,j;
  i=0;
  j=0;
  printf("1°) Je suis ici\n");
  printf("i=%d j=%d\n",i,j);
  if ((i==0) && (i=j)) {
    /* (i=j) => i vaudra 0, et 0 est identique à faux !!! */
    printf("2°) Je suis ici\n");
    ...
  }
  return 0 ;
}
```

## 11.5 Bonne chasse...



**Exercice n°11.1** — *Erreur moyenne*

Copiez le programme suivant, puis corrigez-le.

```
#include <stdio.h>
int main () {
  int i, somme;
  for (i=0 ; i<10 ; i++);
    printf ("i=%d\n",i) ;
    somme += i;
  printf("La moyenne vaut:%d",somme/i) ;
  return 0 ;
}
```

Ce programme est censé afficher ceci à l'écran :

```
i=0
i=1
...
i=9
La moyenne vaut: 4.50000
```

## 11.6 Erreurs d'exécution : les erreurs de segmentation…

Ce type d'erreur apparaît lorsque votre programme accède à une zone de la mémoire qui lui est interdite : vous êtes sur un segment de la mémoire [1] sur lequel vous n'avez pas le droit de travailler.

Le programme suivant peut provoquer de telles erreurs :

```
#include <stdio.h>
int main () {
  int i=0;
  scanf("%d",i);

  return 0 ;
}
```

Soit vous voyez tout de suite l'erreur… soit vous ajoutez des mouchards :

```
#include <stdio.h>
int main () {
  int i=0;
  printf("1°) Je suis ici\n") ;

  scanf("%d",i);
  printf("2°) Je suis ici\n");

  return 0 ;
}
```

Ces mouchards vous permettront rapidement de voir que le problème provient de la ligne `scanf("%d",i)` car seul le message « 1°) Je suis ici » s'affiche et pas le message « 2°) Je suis ici » [2].

Le problème vient donc de i qui vaut `0`… le `scanf` va tenter de stocker ce que vous venez d'entrer au clavier à l'adresse mémoire `0` (`NULL`) ! Cette dernière est réservée au système d'exploitation, d'où l'erreur…

Il en va de même du programme ci-dessous qui **pourrait** poser des problèmes du fait que l'on risque de sortir des bornes du tableau.

1. Segment de mémoire (d'après Wikipédia) : espace d'adressage indépendant défini par deux valeurs : l'adresse où il commence et sa taille.
2. N'oubliez pas les `\n` dans vos `printf` pour la raison évoquée plus haut…

```
#include <stdio.h>
#define TAILLE 10

int main () {
  int tab[TAILLE];

  tab[TAILLE+10]=100;

  return 0 ;
}
```

Ce programme peut planter ou non. Ce type d'erreur conduit le plus souvent à des bugs aléatoires, les plus difficiles à corriger !

### 11.6.1 Le debugger ddd

Ce debugger est très efficace pour trouver les erreurs de segmentation.



Copiez, compilez, exécutez le programme suivant.

```
#include <stdio.h>
#include <string.h>
int main () {
  int * p=NULL;
  *p=123;
  printf("\n  je suis ici...\n");
}
```

Le programme ne fonctionne pas et provoque une erreur de segmentation. Nous allons donc le débugger avec `ddd`.



Faites :

- `gcc -o essai essai.c -g`
- `ddd essai`
- fermez les petites fenêtres « parasites » qui apparaissent au lancement de `ddd`
- cliquez sur le bouton **run** (il faut parfois chercher un peu dans les menus)...



L'option `-g` de la ligne de compilation permet de compiler le programme en y incluant les informations supplémentaires utiles au débogage.

Lorsque vous faites `p=NULL;`, vous placez donc la valeur `0` dans cette variable. Ceci signifie que `p` pointe sur un élément mémoire qui n'est pas accessible par votre programme en écriture. Or, vous faites `*p=123;` qui revient à vouloir écrire la valeur `123` à l'adresse `NULL`.

Le debugger `ddd` vous indique alors quelle ligne a provoqué l'erreur de segmentation. Sur un programme de plusieurs centaines, voire plusieurs milliers de lignes, cette aide est particulièrement appréciable.

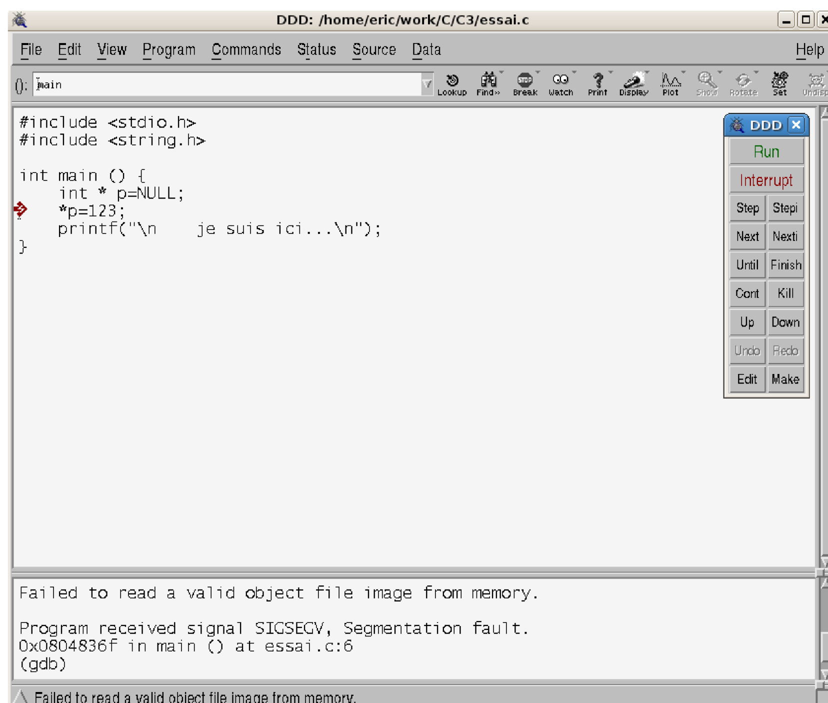



FIGURE 11.1 - Débogage d'un programme



Moralité : en cas d'erreur de segmentation, tentez tout d'abord un `ddd`...

## 11.6.2 Une autre chasse…



**Exercice n°11.2** — *Verlan*

Soit le programme suivant qui doit afficher la chaîne de caractères `chaine` à l'envers, et ce, caractère par caractère :

```
#include <stdio.h>
#include <string.h>

int main () {
  int i;
  char chaine[]="! éuggubéd tse emmargorp el";

  for (i=strlen(chaine); i =! 0; i--)
   printf("%s",chaine[i]) ;
  return 0 ;
}
```

 Corrigez le programme précédent.

### 11.6.3 Dernière sournoiserie…

Testez le programme suivant :

```
#include <stdio.h>

int main () {
int i;
  int i1,i2 ;
  char c1,c2;

  printf("1) Entrez un nombre: ");
  scanf("%d",&i1);
  printf("2) Entrez un nombre: ");
  scanf("%d",&i2);

  printf("1) Entrez une lettre: ");
  scanf("%c",&c1);
  printf("2) Entrez une lettre: ");
  scanf("%c",&c2);

  printf("1) J'ai récupéré lettre 1:%d\n",(int)c1);
  // renverra 10 c.à.d le code ascii de '\n'
  printf("2) J'ai récupéré lettre 2:%d\n",(int)c2);
}
```

On voit que l'`\n` subsiste du premier caractère et pollue la variable `c2`. En effet, quand vous faites un :

```
scanf("%c",&c1);
```

vous demandez à l'ordinateur de lire un unique caractère. Du coup l'`\n` restera de côté (pour l'instant).

Une solution pour remédier à ce travers consiste à utiliser systématiquement la fonction `gets(chaine)` à la place de chaque `scanf` de la façon suivante.

```
/* ===== Version non corrigée ===== */
int i;
char c;
printf("Entrez un caractère:");
scanf("%c",&c);

printf("Entrez un chiffre:");
scanf("%d",&i);
```

```
/* ===== Version corrigée ===== */
int i;
char c;
char chaine[100];

printf("Entrez un caractère:");
gets(chaine);
c=chaine[0];

printf("Entrez un chiffre:");
```

```
gets(chaine);
sscanf(chaine,"%d",&i) ;
```



Cette solution n'est en fait pas viable d'un point de vue sécurité du code (vous obtenez d'ailleurs peut-être un warning). En effet que se passera-t-il quand, au lieu de saisir un caractère, vous en saisirez 200 ? Le tampon de 100 caractères sera dépassé et un problème apparaîtra. Nous n'aborderons pas ici le problème de la saisie sécurisée, mais soyez tout de même conscient(e) du problème potentiel…

## 11.7 Solutions



**Corrigé de l'exercice n°11.1** — *Erreur moyenne*

```
#include <stdio.h>
int main () {
  int i, somme=0;
  for (i=0 ; i<10 ; i++) {
    printf("i=%d\n",i) ;
    somme = somme + i;
  }
  printf("La moyenne vaut:%f",(float) somme/i) ;
  return 0 ;
}
```



**Corrigé de l'exercice n°11.2** — *Verlan*

```
#include <stdio.h>
#include <string.h>

int main () {
  int i;
  char chaine[]="! éuggubéd tse emmargorp el";

  for (i=strlen(chaine); i != 0; i--)
    printf("%c",chaine[i]) ;
  return 0 ;
}
```

## 11.8 À retenir

On retiendra que pour trouver un bug, la méthode est toujours la même :

1. on tente de le reproduire à tous les coups et on note la séquence,
2. on cherche à isoler le plus précisément possible l'endroit où le problème est apparu en injectant des mouchards (trace),
3. dans le cas d'une erreur de segmentation, on tente d'utiliser un debugger.

### 11.8.1 Le debugger ddd

Pour utiliser `ddd` :

- Compilez le programme avec l'option `-g` selon : `gcc -o programme programme.c -g`
- Exécutez : `ddd programme`

# Chapitre 12

# Compléments

## 12.1 Objectif

Ce chapitre vous propose d'aborder quelques notions plus complexes qu'il est nécessaire de comprendre mais où seule la pratique sera votre véritable alliée.

## 12.2 Conversions de type

Pour convertir un type en un autre, on réalise ce que l'on appelle un « cast ». Imaginons que nous souhaitions convertir un nombre du type `float` en un entier du type `int`. Voici comment procéder :

```
int main () {
  float f;
  int i;

  f=3.1415;
  i=(int) f; /* résultat dans i : 3 */
  /* donc la partie décimale est perdue... */
```

```
  return 0 ;
}
```



Sans la conversion explicite, le programme aurait donné la même chose : un `float` est converti en `int` avant d'être stocké dans une variable de type `int` (le compilateur pourrait néanmoins émettre un avertissement).

## 12.3 Usage très utile des conversions de type

Considérons le programme suivant [1] :

```
int main () {
  printf("Résultat : %f",3/4);
  return 0 ;
}
```

Celui-ci affichera `Résultat : 0.0` ! ! !

En effet, l'opérateur « / » réalise une division entière de l'entier `3` par l'entier `4` ce qui vaut `0` (les deux opérandes étant entières, le résultat est converti automatiquement en entier).

Il existe au moins deux solutions pour remédier à ce problème :

1. écrire `3.0` à la place de `3` ce qui va forcer le programme à faire une division en flottants :

```
#include <stdio.h>

int main () {
  printf("Résultat : %f",3.0/4);
  return 0 ;
}
```

2. convertir le nombre `3` en un flottant :

```
#include <stdio.h>

int main () {
  printf("Résultat : %f",(float)3/4);
  return 0 ;
}
```

La division se fera alors en flottants [2].

1. Là aussi, vous pourriez avoir un avertissement du compilateur.

2. L'opérateur de cast `(<type>)` (où `<type>` est un type quelconque) est prioritaire sur les opérateurs binaires. L'expression `(float)3/4` est donc évaluée comme `((float)3)/4` et non comme `(float)(3/4)`.



On notera que la solution suivante ne fonctionnerait pas :

```
#include <stdio.h>

int main () {
 printf("Résultat : %f",(float)(3/4));
 return 0 ;
}
```

En effet, nous réalisons tout d'abord la division de `3` par `4` (avec pour résultat l'entier `0`), puis la conversion de ce dernier en flottant.

Le résultat est donc :

**`Résultat : 0.0`**

## 12.4 Fonction putchar

Le programme suivant :

```
#include <stdio.h>

int main () {
 char c='A';
 putchar(c);

 return 0 ;
}
```

fait la même chose que :

```
#include <stdio.h>

int main () {
 char c='A';
 printf("%c",c);
 return 0 ;
}
```

Nous constatons que `putchar` affiche un unique caractère à l'écran.

## 12.5 Allocation dynamique de mémoire

### 12.5.1 Fonctions malloc et sizeof

La fonction `malloc`, déclarée dans `stdlib.h` permet de réserver un bloc mémoire au cours de l'exécution du programme.

Ainsi, `malloc(N)` fournit l'adresse d'un bloc en mémoire de `N` octets libres ou la valeur `NULL` (c'est à dire 0) s'il n'y a pas assez de mémoire.

Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de moins de 4000 caractères.

1. nous créons un pointeur `tab` vers un caractère (`char *tab`).
2. nous exécutons l'instruction : `tab = malloc(4000);` qui fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à `tab`.

S'il n'y a plus assez de mémoire, c'est la valeur `NULL` (c'est à dire 0) qui est affectée à `tab`.

Si nous souhaitons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine / d'un système à l'autre, nous aurons besoin de la grandeur effective d'une donnée de ce type. Il faut alors utiliser l'opérateur `sizeof` afin de préserver la portabilité du programme :

- `sizeof(<var>)` : fournit la taille, en octets, de la variable `<var>`
- `sizeof(<constante>)` : fournit la taille de la constante `<const>`
- `sizeof(<type>)` : fournit la taille pour un objet du type `<type>`

Exemple :

Soit la déclaration des variables suivantes :

```
short tab1[10];
char tab2[5][10];
```

| Instructions | Valeurs retournées | Remarques |
|---|---|---|
| **sizeof(tab1)** | 20 | |
| **sizeof(tab2)** | 50 | |
| **sizeof(double)** | 8 | Généralement ! |
| **sizeof("bonjour")** | 8 | Pensez au zéro final des chaînes |
| **sizeof(float)** | 4 | Généralement ! |

TABLE 12.1 - Déclaration de variables



Si nous souhaitons réserver de la mémoire pour `x` valeurs de type `int`, la valeur de `x` étant lue au clavier :

```
int x;
int *pNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &x);
pNum = malloc(x*sizeof(int));
```

Certains compilateurs imposent d'écrire

```
pNum = (int *) malloc(x*sizeof(int));
```

En effet, étant donné que `malloc` renvoie un pointeur quelconque (`void *`), nous devons convertir cette adresse par un *cast* en un pointeur sur un entier, d'où l'usage de `(int *)`.

Voici d'autres exemples :

```
char * pointeur_sur_chaine;
char * pointeur_sur_float;

pointeur_sur_chaine = (char *) malloc(1000*sizeof(char));
pointeur_sur_float = (float *) malloc(10000*sizeof(float));
```



Le programme suivant lit 20 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe les adresses aux composantes du tableau `phrases[]`. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et s'interrompt avec le code d'erreur -1. Nous devons utiliser une variable d'aide `phrase` comme zone intermédiaire (non dynamique). Pour cette raison, la longueur maximale d'une phrase est fixée à 100 caractères[1].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LONGUEUR_MAX_PHRASE 100 /* longueur maximum d'une phrase */
#define NOMBRE_MAX_PHRASES 20 /* nombre maximum de phrases */

int main() {

  char phrase[LONGUEUR_MAX_PHRASE];
  char *phrases[NOMBRE_MAX_PHRASES];

  int i;

  for (i=0; i<NOMBRE_MAX_PHRASES; i++) {
   printf("Entrez une phrase SVP...");
   gets(phrase);

   /* Réservation de la mémoire */
   phrases[i] = (char *) malloc(strlen(phrase)+1);

   /* S' il y a assez de mémoire, ... */
   if (phrases[i]!=NULL) {
    /* copier la phrase à l'adresse renvoyée par malloc, ... */
    strcpy(phrases[i],phrase);
   }
   else {
    /* sinon faire une sortie "catastrophe" */
    printf("Attention! Plus assez place en mémoire !!! \n");
    exit(-1);
   }
  }

  return 0;
}
```

1. À la fin de ce programme, nous devrions libérer la mémoire (voir paragraphes suivants).

### 12.5.2 Fonction free

Lorsque nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de `malloc`, nous pouvons le libérer à l'aide de la fonction `free` déclarée dans `<stdlib.h>`.

L'appel à `free(<Pointeur>)` libère le bloc de mémoire désigné par `<Pointeur>`. L'appel à la procédure n'a pas d'effet si le pointeur passé en paramètre possède la valeur zéro.



– La fonction `free` peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par `malloc`.
– La fonction `free` ne change pas le contenu du pointeur ; il est conseillé d'affecter la valeur `NULL` au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
– Si nous ne libérons pas explicitement la mémoire à l'aide de `free`, celle-ci peut être libérée automatiquement à la fin du programme (mais cette façon de faire est à éviter). Ce processus de libération de la mémoire dépend du système d'exploitation utilisé. Une telle fuite de mémoire s'appelle « Memory Leak ».



Par exemple :

```
char * pointeur_sur_chaine;
pointeur_sur_chaine = (char *) malloc(1000*sizeof(char));
...

/* à présent, nous n'avons plus besoin de cette zone mémoire : */
free(pointeur_sur_chaine);
pointeur_sur_chaine=NULL;
```

## 12.6 Avez-vous bien compris ceci ?

Considérons le programme suivant :

```
1. #include <stdio.h>
2.
3. int main() {
4.   int i;
5.   int * p;
6.   printf ("%d\n",i);
7.   *p = 12;
8.   return 0;
9. }
```

– À la ligne 6, le programme affichera n'importe quoi car la variable `i` n'a pas été initialisée.
– La ligne 7 a de fortes chances de faire bugger le programme ! En effet `p` n'a pas été initialisé. En particulier si on faisait un `printf("%p",p)`, on pourrait constater que sa valeur est indéfinie. En effet, `p`, n'est qu'un pointeur, c'est-à-dire une variable contenant une valeur. Donc `p` désigne une adresse quelconque qui peut être celle d'une autre variable. Ainsi, le bug sera dû au fait que `p` pointe n'importe où et que vous essayez d'initialiser ce n'importe où (qui ne vous appartient peut-être pas).

# 12.7 Sur l'utilité des pointeurs

## 12.7.1 Modifier une variable dans une fonction

Vous savez qu'une fonction n'est pas capable de modifier ses arguments :

```
#include <stdio.h>

void calcule_double (int x){
  x=x+x ;
}

main () {
  int i=2;
  calcule_double(i);
  printf("i vaut à présent :%d",i);/* i vaut toujours 2 !!! */
}
```

La solution consiste donc à faire (voir le déroulé plus loin).

```
#include <stdio.h>

void calcule_double (int * pointeur_int){
  * pointeur_int = (* pointeur_int) + (* pointeur_int) ;
}

main () {
  int i=2;
  calcule_double(&i);
  printf("i vaut à présent :%d",i); /* i vaut bien 4 !!! */
}
```

Imaginons que `pointeur_int` se trouve en mémoire logé à l'adresse `20` et `i` à l'adresse `10` :

| | | |
|---|---|---|
| Nom de la variable | **i** | **pointeur_int** |
| Contenu | | |
| Position en mémoire | 10 | 20 |

Figure 12.1 - Adresse mémoire (a)

À présent, déroulons le programme principal :

1. `int i=2` : *déclaration et initialisation de i à 2*
2. `calcule_double(&i)` : *appel de la fonction calcule_double, avec la valeur 10 (l'adresse de i)*
3. `void calcule_double (int * pointeur_int)` : *on lance cette fonction et pointeur_int vaut donc 10*
4. `* pointeur_int = (* pointeur_int)+(* pointeur_int)` : *(* pointeur_int) pointe sur i. L'ancienne valeur de* `i` *va être écrasée par la nouvelle valeur : 2+2*
5. `printf("i vaut à présent : %d", i)` : *on se retrouve avec 4 comme valeur de i.*

| | | |
|---|---|---|
| Nom de la variable | `i` | `pointeur_int` |
| Contenu | 2 | 10 |
| Position en mémoire | 10 | 20 |

FIGURE 12.2 - Adresse mémoire (b)

### 12.7.2 Saisie d'un nombre dans une fonction

Étudions le programme suivant dont le but est simplement de modifier la valeur de `n` :

```
#include <stdio.h>

void saisie (int *pointeur);

int main() {
  int n;
  saisie(&n);
  printf("n vaut : %d",n);

  return 0;
}

void saisie (int *pointeur) {
  printf("Entrez un nombre :");
  scanf ("%d",pointeur);
}
```

Imaginons que `pointeur` se trouve en mémoire logé à l'adresse `100` et `n` à l'adresse `1000` :

| | | |
|---|---|---|
| Nom de la variable | `pointeur` | `n` |
| Contenu | | |
| Position en mémoire | 100 | 1000 |

FIGURE 12.3 - Adresse mémoire (c)

À présent, déroulons le programme :

1. `int n` : *déclaration de n. Il vaut n'importe quoi vu qu'il n'a pas été initialisé !*
2. `saisie(&n)` : *appel de la fonction saisie, avec la valeur 1000 (l'adresse de n)*
3. `void saisie (int *pointeur)` : *pointeur vaut donc 1000*
4. `printf("Entrez un nombre :")`
5. `scanf("%d",pointeur)` : *la fonction scanf va stocker ce que l'utilisateur tape au clavier à partir de l'adresse mémoire 1000* (imaginons que l'on entre la valeur 42).
6. retour dans `main`, la valeur entrée (42) a été stockée à l'adresse 1000, donc dans `n`. Le programme affichera `n vaut : 42`

## 12.8 Un mot sur les warnings

Lorsque vous compilez, vous pouvez obtenir des erreurs, des *warnings* (avertissements) ou parfois les deux.

Les erreurs, vous connaissez : lorsque écrivez par exemple `floatt` au lieu de `float`, c'est une erreur. S'il y a des erreurs à la compilation (`gcc -o essai essai.c`), le compilateur ne génèrera pas de fichier exécutable (fichier `essai`). En revanche, s'il n'y a que des *warnings*, le fichier `essai` sera créé, mais le compilateur nous alerte au sujet d'une erreur possible.

Il est important de comprendre qu'en langage C, un *warning* équivaut à une « erreur larvée ». Ainsi, vous pouvez effectivement exécuter votre programme et ce malgré les warnings mais le problème est que vous risquez de le « payer » par la suite :



```
#include <stdio.h>
#include <stdlib.h>

void affiche_matrice(int matrice[9][9]) {
  int i,j;

  for(i=1; i<=7; i++) {
    for(j=1; j<=7; j++)
      printf("|%d", matrice[i][j]);
    printf("|\n");
  }
}

int main() {
  int i;
  int matrice[9][9];
  ...
  affiche_matrice(matrice[9][9]);
}
```

La dernière ligne `affiche_matrice(matrice[9][9])` vous renverra un warning… Pourquoi ? Prenons le cas où cette ligne serait remplacée par :

```
affiche_matrice(matrice[1][1]);
```

Dès lors, on voit bien le problème, cette ligne ne passe pas tout le tableau `matrice` à la fonction `affiche_matrice`, mais uniquement la case `[1][1]`. La solution consisterait donc à écrire :

```
affiche_matrice(matrice);
```

En conclusion, vous devez considérer tous les warnings comme des erreurs et les éliminer (cela ne concerne cependant pas les warnings provenant de l'usage de `gets` même si cette considération dépend de l'usage qui en est fait au sein de votre programme).

# Chapitre 13

# Quelques exemples de programmes

## 13.1 Objectifs

Le but de ce chapitre est de vous montrer quelques problèmes accompagnés de leur solution. On apprend beaucoup de choses en lisant des programmes finis ;-)

## 13.2 Convertisseur francs/euros

Voici un programme qui produit une petite table de conversion francs/euros.

```
#include <stdio.h>
#define TAUX 6.55957

int main () {
 float francs;

 francs=0;
 while (francs<=10) {
  printf("%4.1f francs = %.2f euros\n",francs,francs/TAUX);
  francs=francs+0.5;
```

```
  }

  return 0;
}
```

L'exécution donnera :

```
 0.0 francs = 0.00 euros
 0.5 francs = 0.08 euros
 1.0 francs = 0.15 euros
 1.5 francs = 0.23 euros
 2.0 francs = 0.30 euros
 2.5 francs = 0.38 euros
 3.0 francs = 0.46 euros
 3.5 francs = 0.53 euros
 4.0 francs = 0.61 euros
 4.5 francs = 0.69 euros
 5.0 francs = 0.76 euros
 5.5 francs = 0.84 euros
 6.0 francs = 0.91 euros
 6.5 francs = 0.99 euros
 7.0 francs = 1.07 euros
 7.5 francs = 1.14 euros
 8.0 francs = 1.22 euros
 8.5 francs = 1.30 euros
 9.0 francs = 1.37 euros
 9.5 francs = 1.45 euros
10.0 francs = 1.52 euros
```

Notez l'utilisation du format dans le `printf` :

- `%4.1f` signifie : écrire en utilisant 4 caractères, dont 1 après la virgule exactement (si l'écriture du nombre nécessite moins de 4 caractères, l'affichage sera calé à droite).
- `%.2f` signifie : écrire le nombre en affichant 2 chiffres après la virgule exactement.

## 13.3 Proportion de nombres pairs et impairs

Voici un programme dont le but est de générer 1000 nombres pseudo-aléatoires et de compter combien sont pairs. En générant suffisamment de nombres, nous devrions nous rapprocher de 50%

```
#include <stdio.h>
#include <stdlib.h> /* pour le générateur pseudo-aléatoire */
#include <time.h>

int main () {
  int nb_hasard=0;
  int i;

  int nb_pairs=0; /*compte le nombre de nombres pairs */
  int nb_impairs=0; /*compte le nombre de nombre impairs */

  srand (time (NULL)); /*initialisation du générateur
      de nombres aléatoires à une valeur
      différente à chaque exécution */
```

```
  i=0;
  do {
    nb_hasard = rand ();
    if (nb_hasard % 2==0) /* c'est un nombre pair */
      nb_pairs=nb_pairs+1;
    else
      nb_impairs=nb_impairs+1;
    i++;
  }
  while (i<1000);

  printf("Proportion de nombres pairs=%f\n",(float)nb_pairs/i);
  printf("Proportion de nombres impairs=%f\n",(float)nb_impairs/i);
  return 0;
}
```



Notez l'utilisation du *cast* dans les dernières lignes. Les deux variables `nb_pairs` et `i` étant entières, la division aurait été arrondie (toujours à 0 en l'occurrence). La conversion de type `(float)` est donc ici très importante.

## 13.4 Affichage d'une table de multiplication

Nous souhaitons obtenir la table de multiplication suivante :

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Voici le programme qui réalise cette tâche :

```
#include <stdio.h>

int main () {
  int ligne, colonne;

  for (ligne=1;ligne<=10;ligne++) {
    for (colonne=1;colonne<=10;colonne++) {
      printf("%4d",ligne*colonne); /* affichage sur 4 caractères */
    }
    printf("\n");
  }

  return 0;
}
```

## 13.5 Maximum d'un tableau

Voici un exemple de fonction qui renvoie le maximum d'un tableau qui lui est passé en paramètre. Voir le commentaire en dessous.

```
#include <stdio.h>

int max_Tableau (int tab[], int taille);

int main() {
  int t1[] = {1,10,4,5,-7}, t2[] = {2,1,14,3} ;
  printf("Maximum de t1 : %d\n", max_Tableau(t1,5) );
  printf("Maximum de t2 : %d\n", max_Tableau(t2,4) );
  return 0;
}

int max_Tableau (int tab [], int taille) {
  int i, max;
  for (i=1, max=tab[0]; i< taille; i++) {
    if (max<tab[i]) max=tab[i];
  }
  return max;
}
```



En fait, seule l'adresse de la 1ère case du tableau est passée en paramètre à la fonction `max_Tableau`



La ligne qui suit le `#include` est la *déclaration* de la fonction `max_Tableau`. En effet, l'appel à la fonction (dans `main`) figure avant la définition de la fonction. Dans une telle situation, il est nécessaire d'annoncer au compilateur l'existence de la fonction en précisant le type de variables en paramètres, et le type de variable renvoyé.

## 13.6 Inverser les éléments d'un tableau

*Remarque préliminaire :* Les deux programmes suivants ont des effets rigoureusement équivalents :

```
#include <stdio.h>
#define TAILLE 10

int main () {
  int i,j;

  for (i=0, j=TAILLE-1 ; i< j; i++,j--)
    printf("i=%d j=%d\n",i,j);
  return 0;
}
```

```
#include <stdio.h>
#define TAILLE 10

int main () {
  int i,j;
```

```
  i=0;
  j=TAILLE-1;

  while (i<j) {
    printf("i=%d j=%d\n",i,j);
    i++, j--;
  }
  return 0;
}
```

Voici à présent un programme qui remplit un tableau de 10 cases par des valeurs saisies au clavier. Dans un second temps, le programme inverse l'ordre des éléments du tableau. Ainsi, si au départ nous avions les dix valeurs suivantes dans le tableau : `1, 2, 3, 4, 5, 6, 7, 8, 9, 10` ; à l'arrivée, nous aurons le tableau suivant : `10, 9, 8, 7, 6, 5, 4, 3, 2, 1`.

L'idée est d'échanger les éléments du tableau à l'aide de deux indices qui parcourent le tableau en commençant respectivement au début et à la fin du tableau et qui se rencontrent en son milieu. Tandis que l'indice `i` va démarrer au début du tableau, l'indice `j` va démarrer à la fin. Lorsque les indices se croisent, on arrête.

Dès que les programmes sont un peu longs, il est préférable de décomposer le problème sous forme de fonctions.

```
#include <stdio.h>
#define TAILLE 10

/* Procédure permettant la saisie d'un tableau de taille n */
void saisie(int t[], int n){
  int i;
  for (i=0; i<n; i++) {
    printf("Elément %d: ", i+1);
    scanf("%d", &t[i]);
  }
}
/*Procédure affichant un tableau de taille n */
void affiche (int t[],int n) {
  int i;
  for (i=0; i<n; i++) {
    printf ("%d ", t[i]);
  }
  printf("\n");
}

/* Procédure inversant le tableau */
void miroir (int t[], int n) {
  int i,j; /* indices courants */
  int aide; /* pour l'échange */

  for (i=0, j=n -1; i< j; i++, j--) {
    /* Echange de t[i] et t[j] */
    aide = t[i];
    t[i] = t[j];
    t[j] = aide;
  }
}

int main() {

  /* Déclarations */
  int tab[TAILLE]; /* tableau donné */
```

```
 saisie (tab,TAILLE);

 printf("Tableau donné : \n");
 affiche(tab,TAILLE);

 miroir(tab,TAILLE);

 printf("Tableau résultat:\n");
 affiche(tab,TAILLE);

 return 0;
}
```

## 13.7 Tri d'un tableau

Supposons que l'on dispose d'un tableau `tab` qui contient `10` valeurs. Nous souhaitons trier ce tableau. Une solution toute simple consiste à faire un passage sur le tableau et à comparer la case d'indice `n` avec celle d'indice `n+1`.

Si la case se trouvant à l'indice `n` contient une valeur plus grande que celle de l'indice `n+1`, alors on inverse les deux valeurs, et ainsi de suite. Voici ci-dessous un exemple sur un tableau de 10 cases.

Tableau initial :

| Indice de la case : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Valeur stockée : | 12 | 10 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 |

Table 13.1 - Tri d'un tableau (1)

On teste la case d'indice `0` et la case d'indice `1`, si besoin est, on permute :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **12** | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 |

Table 13.2 - Tri d'un tableau (2)

On teste la case d'indice `1` et la case d'indice `2`, si besoin est, on permute :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | **4** | **12** | 5 | 6 | 7 | 8 | 9 | 10 | 1 |

Table 13.3 - Tri d'un tableau (3)

...

Au bout d'un parcours complet du tableau, on obtient :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 12 |

TABLE 13.4 - Tri d'un tableau (4)

Nous constatons que le tableau est « mieux » trié, mais ça n'est pas encore parfait. Dans le pire des cas (tableau trié dans l'ordre décroissant) `n-1` passages seront nécessaires et suffisants pour trier un tableau de taille `n`.

Cette méthode de tri porte un nom, il s'agit du *tri à bulles*.

Voici l'ensemble des fonctions qui réalisent ce travail :

```
#include <stdio.h>
#define TAILLE 10

/******************************************/
void saisie(int t[], int n);
void affiche(int t[],int n);
void tri_tableau (int tab[], int taille);
/******************************************/

int main () {
 int tab[TAILLE];
 saisie(tab,TAILLE);
 tri_tableau(tab,TAILLE);
 printf("Voici votre tableau trié :\n");
 affiche(tab,TAILLE);
 return 0;
}

/* Procédure permettant la saisie d'un tableau de taille n */
void saisie(int t[], int n){
 int i;
 for (i=0; i<n; i++) {
  printf("Elément %d : ", i+1);
  scanf("%d",& t[i]);
 }
}

/* Procédure de tri */
void tri_tableau (int tab[], int taille) {
 int i,j;
 int temp;
 /* tri du tableau */
 for (i=0; i<taille-1; i++)
  for (j=0; j<taille-1; j++)
   if (tab[j]>tab[j+1]) { /* échange de valeurs */
    temp=tab[j];
    tab[j]=tab[j+1];
    tab[j+1]=temp;
   }
}

/* Procédure affichant un tableau de taille n */
void affiche(int t[],int n){
 int i;
 for (i=0; i<n; i++) {
  printf("%d ", t[i]);
```

```
  }

  printf("\n");
}
```

Il est important d'avoir bien tout compris dans ce programme…



Notez que dans ce programme, nous avons repris des fonctions qui avaient été écrites pour un autre problème (`saisie` et `affiche` écrites pour le programme qui affiche un tableau à l'envers). Un programmeur aime beaucoup réutiliser le travail qu'il a déjà fait. Les fonctions servent en partie à cela : à être réutilisables. Rares sont les programmeurs qui n'ont pas leur boîte à outils de fonctions avec eux !

Par ailleurs, le programme de tri peut être amélioré. En effet, dans notre programme, nous effectuons toujours `n-1` passages, qui sont nécessaires pour trier dans le pire des cas. Parfois, le tableau peut être néanmoins trié en moins de passes. La boucle `for` extérieure pourra être avantageusement remplacée par une boucle `while` qui s'arrête si le tableau est effectivement trié (parce qu'il n'y a eu aucun échange de valeurs lors du dernier passage par exemple). Si ce dernier point vous paraît un peu compliqué, vous pourrez y revenir plus tard et y réfléchir à nouveau.

## 13.8 Jeu de la vie

### 13.8.1 Historique

John Horton Conway est un mathématicien qui a exercé à l'Université de Cambridge puis a Princeton. Très prolifique en matière de jeux mathématiques, il décrivit en 1970 le *jeu de la vie*, visant à modéliser d'une façon simplifiée l'évolution d'organismes vivants.

### 13.8.2 Règles du jeu

Le jeu de la vie se joue normalement sur un damier infini. Chaque case est occupée par une cellule qui peut être vivante ou morte. À chaque génération, chaque cellule peut naître, mourir, ou rester dans son état. Les règles qui permettent de passer d'une génération à l'autre sont précises et ont été choisies avec soin pour que l'évolution des organismes soit intéressante et semble imprévisible. En premier lieu, notons que sur un damier infini, chaque case a exactement huit voisins (si on considère aussi les voisins par une diagonale [1]). Les règles du jeu de la vie sont les suivantes :

- une cellule vivante ayant exactement 2 ou 3 voisins vivants survit à la génération suivante.
- une cellule vivante ayant de 4 à 8 cellules voisines vivantes meurt d'étouffement à la génération suivante.
- une cellule vivante ayant zéro ou une cellule voisine vivante meurt d'isolement à la génération suivante.
- sur une case vide ayant exactement 3 voisins vivants, une cellule naîtra à la génération suivante.

1. Ce type de voisinage s'appelle voisinage de Moore.

Notons que c'est l'ensemble de la génération *actuelle* qui doit être pris en compte pour l'établissement de l'état des cellules à la génération *suivante*.

Voici un exemple de figure sur un petit damier. Les cellules qui devront mourir à la génération suivante sont grisées (cette figure porte un nom, il s'agit d'un planeur) :

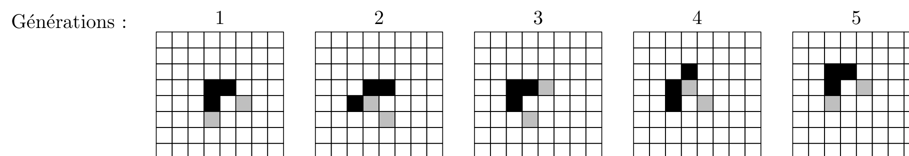



FIGURE 13.1 - Générations (le jeu de la vie)

### 13.8.3 Images obtenues

Si nous faisions une version graphique, voici ce que nous pourrions obtenir (figures 13.2 à 13.3) :



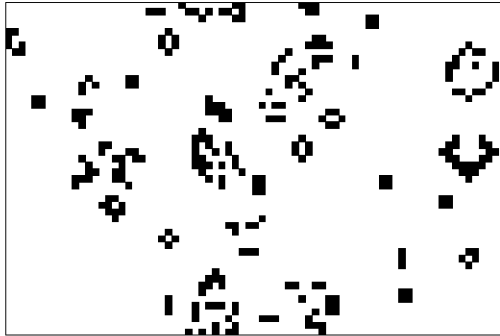
FIGURE 13.2 - Le jeu de la vie - configuration aléatoire de départ

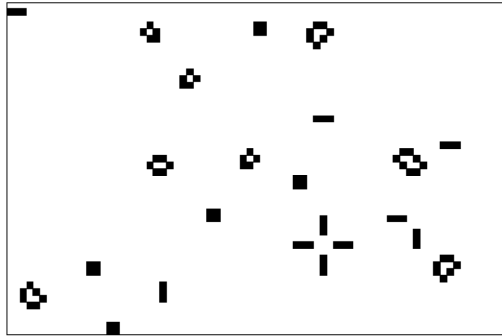
Certaines configurations réapparaissent spontanément sur un damier initialement aléatoire, comme les planeurs mentionnés ci-dessus. Des centaines de figures sont ainsi recensées pour leur « comportement » plus ou moins remarquable (le lecteur intéressé pourra faire des recherches très fructueuses sur Internet).

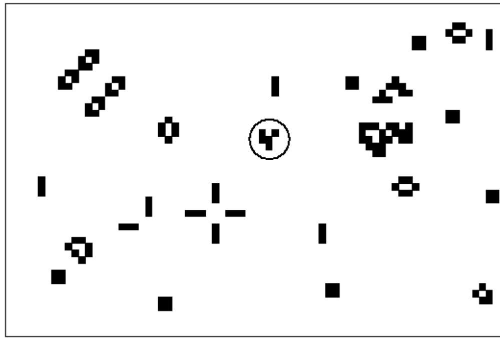
### 13.8.4 Proposition de programme

Nous allons considérer que toutes les cellules sont stockées dans une matrice (notre damier ne sera donc pas infini). Pour une case `m[i][j]`, les huit voisins sont :

```
m[i-1][j], m[i+1][j], m[i][j-1], m[i][j+1]
```


Formations d'amalgames de cellules...


Configuration finale de motifs de période 1 ou 2...


Apparition d'un planeur...

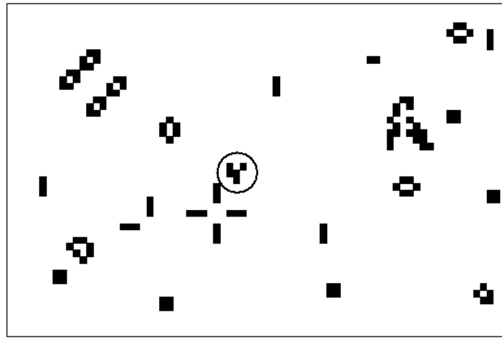

...qui se balade.

FIGURE 13.3 - Le jeu de la vie - suite

```
m[i-1][j-1], m[i+1][j+1], m[i+1][j-1], m[i-1][j+1]
```

Pour éviter les problèmes qui se posent au bord de la matrice[1] nous ne considérerons comme faisant partie du jeu que les cellules qui ne sont pas sur la couronne extérieure de la matrice (voir figure suivante) :

1. Rappelez-vous qu'accéder à un tableau en dehors de ses bornes est une erreur fréquente, et assez difficile à détecter.

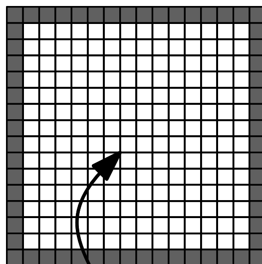



Figure 13.4 - Damier (le jeu de la vie)

Nous considérerons que la couronne extérieure (grisée sur l'image) est constituée de cellules mortes qui ne sont pas mises à jour.

On se propose de découper le programme de la façon suivante (lisez ce programme, qui n'est pas encore complet, ainsi que les explications ci-dessous) :

```
/*************************************
         JEU DE LA VIE
****************************************/

#include <stdio.h>
#include <stdlib.h>
#define TAILLE_SOUS_MATRICE 7
#define TAILLE_SUR_MATRICE 9
/* Taille de la matrice contenant */
/* les cellules + 2 pour la bordure */

/****************************************/
/******* P R O T O T Y P E S ********/
/****************************************/

/* Initialisation de la matrice de départ */
void init(int matrice [][TAILLE_SUR_MATRICE ]);

/* Indique pour la cellule de coordonnées (ligne,colonne) */
/* le nombre de cellules voisines vivantes */
int nombre_voisins (int matrice [][TAILLE_SUR_MATRICE], int ligne, int colonne);

/* Réalise une étape de plus dans la mise à jour de la matrice: */
/* autrement dit, on réalise un cycle de vie */
void mise_a_jour(int matrice[][TAILLE_SUR_MATRICE]);

/* Affichage de la matrice en cours */
void affiche_matrice(int matrice [][TAILLE_SUR_MATRICE]);
/****************************************/
/******* F O N C T I O N S **********/
/****************************************/


int main() {
  int i;
```

```
  int nbr_cycles;
  int matrice[TAILLE_SUR_MATRICE] [TAILLE_SUR_MATRICE ];

}

void init(int matrice [][TAILLE_SUR_MATRICE]) { ... }

void mise_a_jour(int matrice[][TAILLE_SUR_MATRICE]) {
  int matrice_densite[TAILLE_SOUS_MATRICE][ TAILLE_SOUS_MATRICE];
...
}
```

Lisons le prototype de la fonction init :

```
void init(int matrice[] [TAILLE_SUR_MATRICE]);
```

Il aurait été possible d'écrire :

```
void init(int matrice[TAILLE_SUR_MATRICE] [TAILLE_SUR_MATRICE]);
```



Nous pourrions penser à tort que la fonction reçoit la matrice `TAILLE_SUR_MATRICE*TAILLE_SUR_MATRICE` en entrée et qu'à la sortie elle ne sera pas modifiée (passage des paramètres par valeur). Dès lors, la fonction `init` ne servirait à rien.

En fait, c'est l'adresse mémoire de la matrice (`&matrice[0][0]`) qui est passée à la fonction `init`. C'est donc un passage par adresse qui est effectué.

La fonction ne pourra effectivement pas modifier l'adresse de `matrice[0][0]` mais, en revanche, pourra modifier les contenus de `matrice[0][0]`, `matrice[1][0]`...

Avant de voir la correction complète de cet exercice, on notera que lors d'un cycle de vie, on ne doit pas modifier la matrice de vie courante au fur et à mesure, elle doit être modifiée d'un coup. Il est donc nécessaire de passer par 2 étapes :

1. construire une matrice intermédiaire qui contient par exemple le nombre de voisins pour chaque cellule.
2. parcourir cette matrice en une passe et modifier la matrice contenant les cellules vivantes.

Voici donc le programme complet :

```
/***************************************
         JEU DE LA VIE
***************************************/

#include <stdio.h>
#include <stdlib.h>
#define TAILLE_SOUS_MATRICE 7
/* On peut avoir 7 * 7 cellules vivantes */
#define TAILLE_SUR_MATRICE 9
/* On place une bordure autour qui facilite la vie du programmeur... */

/***************** PROTOTYPES ***********************/
void init(int matrice [][TAILLE_SUR_MATRICE ]);
int nombre_voisins (int matrice [][TAILLE_SUR_MATRICE ],
               int ligne, int colonne);
```

```
void mise_a_jour(int matrice[][TAILLE_SUR_MATRICE ]);
void affiche_matrice(int matrice [][TAILLE_SUR_MATRICE ]);
void ligne(int largeur);
/*****************************************************/

int main( ) {
  int i;
  int nbr_cycles;
  int matrice[TAILLE_SUR_MATRICE] [TAILLE_SUR_MATRICE ];
  char s[2];

  printf("Nombre de cycles : ");
  scanf("%i",&nbr_cycles);

  init(matrice);
  printf("La population au départ : \n");
  affiche_matrice(matrice);
  printf("Pressez sur ENTER pour continuer...\n");
  gets(s);

  for(i=0; i<nbr_cycles; i++) {
    mise_a_jour (matrice);
    printf("La population après %d cycles: \n", i+1);
    affiche_matrice (matrice);
    printf("Pressez sur ENTER pour continuer...\n");
    gets(s);
  }
  return 0;
}
```

```
/****************************************/
/* Initialisation de la matrice */
void init(int matrice [][TAILLE_SUR_MATRICE ]) {
/****************************************/
  int i,j;

  for(i=0; i< TAILLE_SUR_MATRICE; i++) {
    for(j=0; j< TAILLE_SUR_MATRICE; j++) {
      if (i<=j && i>0 && j<=7)
        matrice[i][j]=1;
      else
        matrice[i][j]=0;
    }
  }
  /* On pourrait aussi faire une initialisation aléatoire */
}

/****************************************/
/* Calcul du nombre de voisins vivants */
int nombre_voisins (int matrice[][TAILLE_SUR_MATRICE ],
                  int ligne, int colonne) {
/****************************************/
  int compte=0; /* compteur de cellules */
  int i,j;

/* On additionne les 9 cellules centrées en ligne,colonne */

  for (i=ligne-1;i<=ligne+1;i++)
    for(j=colonne-1;j<=colonne+1;j++)
      compte=compte+matrice[i][j];

      /* Puis on retire celle du milieu... */
```

```
    compte -= matrice[ligne][colonne];

    return compte;
}

/****************************************/
/* Correspond à l'étape n+1 */
void mise_a_jour(int matrice[ ][TAILLE_SUR_MATRICE ]) {
/****************************************/
  int i,j;
  int nbr_voisins;
  int matrice_densite[TAILLE_SOUS_MATRICE][TAILLE_SOUS_MATRICE];
  /* matrice qui comptabilise le nombre de voisins */
  /* et cela, case par case */

  for(i=0; i< TAILLE_SOUS_MATRICE; i++)
    for(j=0; j< TAILLE_SOUS_MATRICE; j++)
      matrice_densite[i][j]=nombre_voisins(matrice,i+1,j+1);
   /* i+1 et j+1 car on passe de la SOUS_MATRICE à la MATRICE */

  for(i=0; i< TAILLE_SOUS_MATRICE; i++)
   for(j=0; j< TAILLE_SOUS_MATRICE; j++) {
      nbr_voisins=matrice_densite[i][j];
      if(nbr_voisins==2)
         matrice[i+1][j+1]=1;
      else if (nbr_voisins==0 || nbr_voisins==4)
         matrice[i+1][j+1]=0;
  }
}
```

```
/****************************************/
/* Affichage à l'écran des cellules vivantes */
void affiche_matrice(int matrice[ ][TAILLE_SUR_MATRICE ]) {
/****************************************/
  int i,j;

  for(i=1; i<=TAILLE_SOUS_MATRICE; i++) {
   ligne(7);
   for(j=1; j<= TAILLE_SOUS_MATRICE; j++)
    if (matrice[i][j]==1)
     printf("|%c",'*');
    else
     printf("|%c",'|');
   printf("|\n");
  }
  ligne(TAILLE_SOUS_MATRICE);
}

/****************************************/
/* Tracé d'une ligne */
void ligne(int largeur) {
/****************************************/
  int i;
  for(i=0; i<largeur; i++)
   printf("+-");
  printf("+\n");
}
```

L'exécution de ce programme en mode non-graphique est un peu frustrante. L'adaptation nécessaire pour dessiner réellement les cellules à l'écran est cependant assez simple (fondamen-

talement, il faut uniquement modifier la procédure `affiche_matrice` pour obtenir un affichage différent).

Chapitre

# 14

# En deuxième lecture

## 14.1 Quelques fonctions mathématiques

Pensez à inclure le fichier d'en-tête `math.h` et compilez en tapant :

```
gcc -o essai essai.c -lm
```

Dans le tableau 14.1, vous trouverez quelques fonctions mathématiques très utiles.

## 14.2 Pointeurs et structures

Voici un programme qui utilise un pointeur sur une structure :

```
#include <stdio.h>

typedef struct {
  char nom [40];
  char prenom [20];
  int  age;
} personne;

```

```
int main () {
  personne p;
  personne * pointeur_sur_une_personne;
  printf("Veuillez entrer le nom de la personne:");
  scanf("%s",p.nom);

  printf("Veuillez entrer le prénom de la personne:");
  scanf("%s",p.prenom);

  printf("Veuillez entrer l'âge de la personne:");
  scanf("%d",&p.age); /* ne pas oublier le '&' !!! */

  pointeur_sur_une_personne=&p;
  printf("Voici les caractéristiques de cette personne:\n");
  printf("nom=%s\n",(*pointeur_sur_une_personne).nom);

  /* autre façon de l'écrire */
  printf("nom=%s\n",pointeur_sur_une_personne->nom);

  return 0;
}
```

Notez que cette seconde écriture (plus pratique à l'usage) repose sur une flèche qui est construite avec le signe moins (-) et le signe supérieur (>).

| Fonction | Explication | Exemple |
|---|---|---|
| `exp(x)` | fonction exponentielle | `y=exp(x)` |
| `log(x)` | logarithme naturel | `y=log(x)` |
| `log10(x)` | logarithme à base 10 | `y=log10(x)` |
| `pow(x,y)` | $x^y$ | `z=pow(x,y)` |
| `sqrt(x)` | racine carrée de $x$ | `y=sqrt(x)` |
| `fabs(x)` | valeur absolue | `y=fabs(x)` |
| `floor(x)` | arrondir en moins | `floor(1.9)` renverra `1` |
| `ceil(x)` | arrondir en plus | `ceil(1.4)` renverra `2` |
| `sin(x)` | sinus de $x$ | `y=sin(x)` |
| `cos(x)` | cosinus de $x$ | `y=cos(x)` |
| `tan(x)` | tangente de $x$ | `y=tan(x)` |

TABLE 14.1 - Quelques fonctions mathématiques

## 14.3 En finir avec les warnings du gets

En première lecture, nous pourrions dire que ce type de warning est « normal ». Prenons un exemple :

```
char chaine[10];
gets(chaine) ;
```

Supposons que votre programme soit utilisé sur internet et qu'un utilisateur malveillant entre une chaîne de caractères plus grande que 10 caractères, par exemple « `ABCDEFGHIJKLMNOPQR` ».

Dans ce cas, à l'exécution, votre programme va recopier à partir de l'adresse de la variable `chaine` les caractères `A`, `B`, ... jusqu'à `R`. Dès lors, les zones mémoires qui suivent la variable `chaine` seront écrasées. Ceci explique que le compilateur vous indique un warning.

Pour y remédier, vous pouvez utiliser :

```
char buffer[128];
fgets(buffer, 128, stdin);
```

en sachant que `stdin` (mis pour *standard input* c'est à dire entrée standard) désigne le plus souvent le clavier.



`fgets` présente une particularité que n'avait pas `gets` : elle place un `\n` à la fin de la chaîne qui a été lue (juste avant l'`\0`).

## 14.4 Sortie standard : stdout

Nous venons de voir que `stdin` désignait la plupart du temps le clavier.

De la même façon, `stdout` (mis pour *standard output* c'est à dire sortie standard) désigne le plus souvent l'écran du terminal.

Les deux programmes suivants sont équivalents :

```
int i=123;
printf("%d",i);
```

```
int i=123;
fprintf(stdout,"%d",i);
```

## 14.5 Utilité des prototypes



Rappelez-vous qu'il est plus prudent de placer tous les prototypes de fonctions que vous appelez juste en dessous des `#include`. Un prototype est l'en-tête d'une fonction.



```
#include <stdio.h>

float mystere (int i); // prototype de la fonction mystère
int main () {
  int j;
  printf("Entrez une valeur:");
  scanf("%d",&j);
  printf("Résultat de la fonction mystère:%f\n",mystere(j));
}

float mystere (int i) {
  return i*i;
}
```

Voyons ce qui se produit si nous omettons le prototype :

```
#include <stdio.h>

int main () {
  int j;
  printf("Entrez une valeur:");
  scanf("%d",&j);
  printf("Résultat de la fonction mystere:%f\n",mystere(j));
}

float mystere (int i) {
  return i*i;
}
```

Au moment où le programme rencontre l'appel à la fonction `mystere`, le compilateur découvre pour la première fois cette fonction. Or il ne sait pas ce que fait cette fonction et *a fortiori* il ignore le type du résultat qui sera renvoyé. Dans ce cas, le compilateur suppose (à tort dans notre exemple) que ce résultat sera du type `int` (qui est le type par défaut). Ceci risque de poser des problèmes dans la suite (et provoquera l'émission d'un avertissement) quand le compilateur s'apercevra qu'en fait, la fonction renvoit un `float`.



Le fait que `int` soit le type par défaut explique pourquoi nous pouvons nous permettre d'écrire un `main` comme ceci :

```
main () {
  ...
}
```

plutôt que comme cela :

```
int main () {
  ...
}
```

## 14.6 Opérateur ternaire

Le langage C possède un opérateur ternaire un peu exotique qui peut être utilisé comme alternative à `if - else` et qui a l'avantage de pouvoir être intégré dans une expression. L'expression suivante : `<expr1> ? <expr2> : <expr3>` est traduite comme ceci :

- Si `<expr1>` est non nulle, alors la valeur de `<expr2>` est fournie comme résultat.
- Sinon, c'est la valeur de `<expr3>` qui est fournie comme résultat.



La suite d'instructions :

```
if (a>b)
  maximum=a;
else
  maximum=b;
```

peut être remplacée par :

```
maximum = (a > b) ? a : b;
```

Employé de façon irréfléchie, l'opérateur ternaire peut nuire à la lisibilité d'un programme, mais si nous l'utilisons avec précaution, il fournit des solutions élégantes et concises.



Par exemple :

```
printf("Vous avez %i carte%c \n", n, (n==1) ? ' ' : 's');
```

## 14.7 Tableaux de chaînes de caractères

La déclaration `char mois[12][10]` réserve l'espace mémoire pour 12 mots contenant 10 caractères (dont 9 caractères significatifs, c'est à dire sans `\0`) :

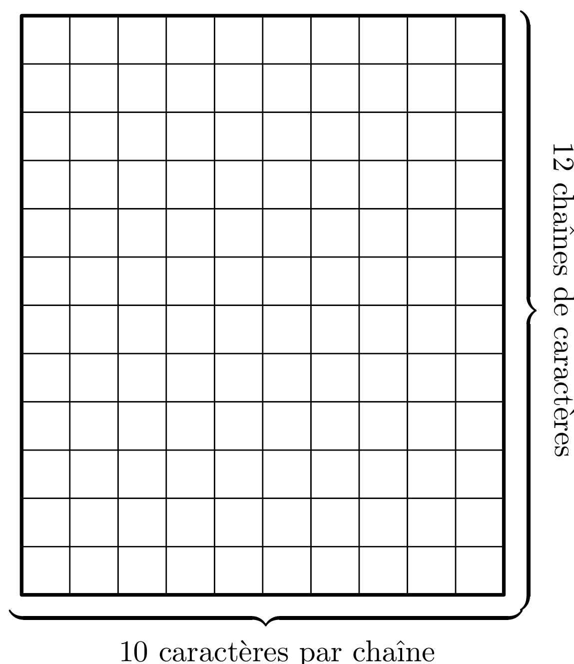



FIGURE 14.1 - Une matrice de caractères

Lors de la déclaration, il est possible d'initialiser toutes les cases du tableau par des chaînes de caractères constantes :

```
char mois[12][10]={"janvier","février","mars","avril","mai","juin",
          "juillet","août","septembre","octobre",
          "novembre","décembre"};
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'j' | 'a' | 'n' | 'v' | 'i' | 'e' | 'r' | '\0' | | |
| 'f' | 'e' | 'v' | 'r' | 'i' | 'e' | 'r' | '\0' | | |
| 'm' | 'a' | 'r' | 's' | '\0' | | | | | |
| 'a' | 'v' | 'r' | 'i' | 'l' | '\0' | | | | |
| 'm' | 'a' | 'i' | '\0' | | | | | | |
| 'j' | 'u' | 'i' | 'n' | '\0' | | | | | |
| 'j' | 'u' | 'i' | 'l' | 'l' | 'e' | 't' | '\0' | | |
| 'a' | 'o' | 'u' | 't' | '\0' | | | | | |
| 's' | 'e' | 'p' | 't' | 'e' | 'm' | 'b' | 'r' | 'e' | '\0' |
| 'o' | 'c' | 't' | 'o' | 'b' | 'r' | 'e' | '\0' | | |
| 'n' | 'o' | 'v' | 'e' | 'm' | 'b' | 'r' | 'e' | '\0' | |
| 'd' | 'e' | 'c' | 'e' | 'm' | 'b' | 'r' | 'e' | '\0' | |

12 chaînes de caractères

10 caractères par chaîne

FIGURE 14.2 - Le contenu du tableau mois



Il est possible d'accéder aux différentes chaînes de caractères d'un tableau, en indiquant simplement la ligne correspondante.



L'exécution des trois instructions suivantes...

```
char mois[12][10]= {"janvier","février","mars","avril",
      "mai","juin","juillet","août","septembre",
      "octobre","novembre","décembre"};
int i = 4;
printf("Aujourd'hui, nous sommes en %s !\n", mois[i]);
```

...affichera, par exemple, la phrase : `Aujourd'hui, nous sommes en mai !` Des expressions comme `mois[i]` représentent l'adresse du premier élément d'une chaîne de caractères.

La remarque précédente est très pratique pour résoudre le problème qui va suivre. Nous allons écrire un programme qui lit un verbe régulier en « er » au clavier et qui affiche la conjugaison au présent de l'indicatif de ce verbe. Nous contrôlerons s'il s'agit bien d'un verbe en « er » avant de conjuguer.



```
Verbe : programmer
je programme
tu programmes
il ou elle programme
nous programmons
vous programmez
ils ou elles programment
```

Nous utiliserons deux tableaux de chaînes de caractères : `sujets` pour les sujets et `terminaisons` pour les terminaisons.

Solution possible :

```c
#include <stdio.h>
#include <string.h>
int main() {
 int i;

 char sujets[6][13] = {"je","tu","il ou elle","nous","vous","ils ou elles"};
 char terminaisons[6][5] = {"e","es","e","ons","ez","ent"};

 char verbe[15]; /* chaîne contenant le verbe */
 int longueur; /* longueur du verbe */

 printf("Quel verbe souhaitez-vous conjuguer ? ");
 scanf("%s", verbe);

 /* S'agit-il d'un verbe se terminant par "er" ? */
 longueur=strlen(verbe);
 if ((verbe[longueur-2] != 'e') || (verbe[longueur-1] != 'r'))
  printf("%s n'est pas un verbe du premier groupe !!!\n",verbe);
 else {
  /* Supprimer la terminaison "er" */
  verbe[longueur-2]='\0';

  /* Conjuguer le verbe */
  for (i=0; i<6; i++)
   printf("%s %s%s\n",sujets[i], verbe, terminaisons[i]);
 }
 return 0;
}
```

## 14.8 Pointeurs et tableaux

Nous allons à présent examiner les liens très étroits qu'il y a entre pointeurs et tableaux. En fait, nous allons voir qu'à chaque fois que vous manipulez des tableaux, comme par exemple à la ligne contenant le `printf` du programme ci-dessous, le langage C va transformer votre instruction `tableau[i]` en se servant de pointeurs…

```c
int tableau[10]={1,2,3,4,5,6,7,8,9,10};

int i;
for (i=0;i<10;i++)
 printf("%d ",tableau[i]);
```

Comme nous l'avons déjà constaté précédemment, le nom d'un tableau représente l'adresse de son premier élément. En d'autres termes : `&tableau[0]` et `tableau` sont une seule et même adresse.

En simplifiant, nous pouvons retenir que le nom d'un tableau est un pointeur constant sur le premier élément du tableau.



En déclarant un tableau `tableau` de type `int` et un pointeur `p` sur `int`,

```
int tableau[10];
int *p;
```

l'instruction : `p = tableau;` est équivalente à `p = &tableau[0];`

Si `p` pointe sur une case quelconque d'un tableau, alors `p+1` pointe sur la case suivante.

Ainsi, après l'instruction : `p = p+1;`

le pointeur `p` pointe sur `tableau[1]`,

| | |
|---|---|
| `*(p+1)` | désigne le contenu de `tableau[1]` |
| `*(p+2)` | désigne le contenu de `tableau[2]` |
| ... | ... |
| `*(p+i)` | désigne le contenu de `tableau[i]` |

Table 14.2 - Pointeurs et tableaux (1)

Il peut sembler surprenant que `p+i` n'adresse pas le `i`ème octet derrière `p`, mais la `i`ème case derrière `p` … Ceci s'explique par la stratégie de programmation des créateurs du langage C : si nous travaillons avec des pointeurs, les erreurs les plus sournoises sont causées par des pointeurs mal placés et des adresses mal calculées. En C, le compilateur peut calculer automatiquement l'adresse de l'élément `p+i` en ajoutant à `p` la taille d'une case multipliée par `i`. Ceci est possible, parce que :

– chaque pointeur accède à un seul type de données ;
– le compilateur connaît le nombre d'octets utilisés pour chaque type.

Enfin, comme `tableau` représente l'adresse de `tableau[0]` :

| | |
|---|---|
| `*(tableau+1)` | désigne le contenu de `tableau[1]` |
| `*(tableau+2)` | désigne le contenu de `tableau[2]` |
| ... | ... |
| `*(tableau+i)` | désigne le contenu de `tableau[i]` |

Table 14.3 - Pointeurs et tableaux (2)

Voici un récapitulatif de tout ceci.

Soit un tableau `tableau` d'un type quelconque et `i` un indice entier alors :

| | | |
|---|---|---|
| `tableau` | désigne l'adresse de | `tableau[0]` |
| `tableau+i` | désigne l'adresse de | `tableau[i]` |
| `*(tableau+i)` | désigne le contenu de | `tableau[i]` |

TABLE 14.4 - Pointeurs et tableaux (3)

Si `p = tableau`, alors :

| | | |
|---|---|---|
| `p` | désigne l'adresse de | `tableau[0]` |
| `p+i` | désigne l'adresse de | `tableau[i]` |
| `*(p+i)` | désigne le contenu de | `tableau[i]` |

TABLE 14.5 - Pointeurs et tableaux (4)



Les deux programmes suivants copient les éléments strictement positifs d'un tableau `tableau` dans un deuxième tableau `positifs`.

Formalisme tableau :

```
int main() {
  int tableau[5] = {-4, 4, 1, 0, -3};
  int positifs[5];
  int i,j; /* indices courants dans tableau et positifs */
  for (i=0,j=0 ; i<5 ; i++)
    if (tableau[i]>0){
      positifs[j] = tableau[i];
      j++;
    }
  return 0;
}
```

Nous pouvons remplacer systématiquement la notation `tableau[i]` par `*(tableau+i)`, ce qui conduit à ce programme qui repose sur le formalisme des pointeurs :

```
int main() {
  int tableau[5] = {-4, 4, 1, 0, -3};
  int positifs[5];
  int i,j; /* indices courants dans tableau et positifs */
  for (i=0,j=0 ; i<5 ; i++)
    if (*(tableau+i)>0){
      *(positifs+j) = *(tableau+i);
      j++;
    }
  return 0;
}
```



Voici un exemple de programme qui range les éléments d'un tableau `tableau` dans l'ordre inverse. Le programme utilise des pointeurs `p1` et `p2` et une variable numérique `aux` pour la permutation des éléments :

```
#include <stdio.h>
#define TAILLE 100

int main() {

  int tableau[TAILLE]; /* tableau donné */
  int dim;          /* nombre réel d'éléments du tableau */
  int aux;          /* variable auxiliaire pour la permutation */
  int *p1, *p2;     /* pointeurs d'aide */
  int i;

  printf("Dimension du tableau (maximum : %d) : ",TAILLE);
  scanf("%d", &dim );

  i=1;
  for (p1=tableau; p1<tableau+dim; p1++) {
    printf("Valeur de l'élément %d : ", i++);
    scanf("%d", p1); // notez l'abscence de '&' !!!
  }

  /* Affichage du tableau avant inversion */
  for (p1=tableau; p1<tableau+dim; p1++)
    printf("%d ", *p1); // ne pas oublier l'étoile !!!
  printf("\n");

  /* Inversion du tableau */
  for (p1=tableau,p2=tableau+(dim-1); p1<p2; p1++,p2--) {
    aux = *p1;
    *p1 = *p2;
    *p2 = aux;
  }
  /* Affichage du résultat */
  for (p1=tableau; p1<tableau+dim; p1++)
    printf("%d ", *p1);
  printf("\n");

  return 0;
}
```

## 14.9 Tableaux de pointeurs

Attention, nous allons compliquer un peu les choses…

Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs.



Déclaration d'un tableau de pointeurs :

`<Type> *<NomTableau>[<N>]`

…déclare un tableau `<NomTableau>` de `<N>` pointeurs sur des données du type `<Type>`.



Par exemple :

```
double * tableau[10];
```

– les crochets [ ] ont une priorité supérieure à l'étoile *
– en lisant de droite à gauche nous voyons que `tableau[10]` sera du type `double *`
– nous déclarons donc un tableau de `10` pointeurs sur des `double`.

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>

int main() {
  char * jours[7];
  int i;
  for (i=0;i<7;i++) {
    jours[i]=(char *) malloc(9);
  }
  strcpy(jours[0],"lundi");
  strcpy(jours[1],"mardi");
  strcpy(jours[2],"mercredi");

  // ...
  return 0;
}
```

Voilà ce que cela donne en pratique :

```
char mois[12][10]= {"janvier","février","mars","avril",
       "mai","juin","juillet","aout","septembre","octobre",
       "novembre","décembre"};
```

déclare un tableau `mois[]` de `12` pointeurs sur `char`. Chacun des pointeurs est initialisé avec l'adresse de l'une des `12` chaînes de caractères (voir figure) :

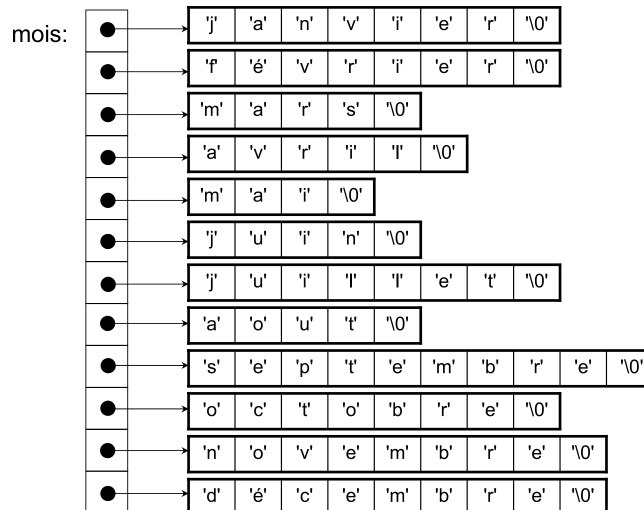



FIGURE 14.3 - – Un tableau de pointeurs sur des chaînes de caractères

Nous pouvons afficher les 12 chaînes de caractères en fournissant les adresses contenues dans le tableau mois à la fonction printf :

```
int i;
for (i=0; i<12; i++)
 printf("%s\n", mois[i]);
```

## 14.10 Choix multiples avec switch

Supposons que vous ayez une cascade de if et else imbriqués à écrire :

```
int i;
printf ("Entrez une valeur:");
scanf("%d",&i);

if (i==0)
 printf ("Nombre nul\n");
else {
 if (i==1)
  printf ("Nombre non égal à un\n");
 else
  printf ("Autre type de nombre\n");
}
```

Dans ce cas, nous pouvons, pour augmenter la lisibilité du programme, utiliser switch/case/break :

```
int i;
printf ("Entrez une valeur:");
scanf("%d",&i);

switch (i) {
 case 0:
  printf ("Nombre nul\n");
  break;
 case 1:
  printf ("Nombre égal à un\n");
  break;
 default:
  printf("Autre type de nombre\n");
  break;
}
```



Notez l'usage du break, en effet, si vous ne faites pas de break, le programme, après être entré dans un case, continuera sur le case suivant et ainsi de suite. Testez le programme suivant en entrant la valeur 0 :

```
int i;
printf ("Entrer une valeur:");
scanf("%d",&i);

switch (i) {
 case 0:
  printf ("Nombre nul\n");
  /* pas de break */
```

```
  case 1:
    printf("Nombre égal à un\n");
    break;
  default:
    printf("Autre type de nombre\n");
    break;
}
```

Le programme affichera à l'exécution :

```
Nombre nul
Nombre égal à un
```

Omettre un `break` n'est pas forcément une faute et cette particularité peut être utilisée pour réaliser des conditions « ou ». Par exemple, la portion de code suivante affichera un message si `a` vaut 1,2 ou 5, un autre message si `a` vaut 3 ou 4 et un troisième message sinon.

```
int a;
printf ("Entrez une valeur:");
scanf("%d",&a);

switch (a) {
  case 1: case 2: case 5:
    printf("Voici le premier message\n");
    break;
  case 3: case 4:
    printf("Voici le second message\n");
    break;
  default:
    printf("Voici le message par défaut\n");
    break;
}
```

## 14.11 Édition de liens

Jusqu'à présent, nous « construisions » nos programmes exécutables en entrant la commande suivante :

```
gcc -o essai essai.c
```

En réalité, cette commande réalise deux opérations : la compilation proprement dite, et l'édition de liens (si la compilation s'est terminée avec succès). Pour réaliser uniquement l'étape de compilation, il faut entrer :

```
gcc -c essai.c
```

Cette étape a pour effet de générer le fichier `essai.o` qui ne contient pour le moment que le code objet qui correspond au source compilé, mais qui ne lie pas les appels de fonctions des bibliothèques extérieures telles que `printf`, `scanf`, `feof`, `sqrt` à leur code respectif.

L'étape suivante, appelée édition de liens, est réalisée en entrant la commande :

```
gcc -o essai essai.o -lm
```

Lors de cette étape, des « références » aux fonctions appelées dans le fichier objet sont ajoutées. Pour cela, les fonctions utilisées sont recherchées automatiquement dans les bibliothèques standard (on y trouvera `printf`, `scanf`…) et dans les bibliothèques spécifiées dans la commande (ici `-lm` qui désigne la bibliothèque mathématique et qui permettra de trouver une référence à la fonction `sqrt` par exemple) [1]

Depuis le début, nous avons employé la syntaxe la plus rapide, et avons lancé la compilation puis l'édition de liens en une seule fois :

```
gcc -o essai essai.c
```

1. Précisons au passage qu'il existe deux sortes d'édition de liens : l'édition de liens dynamiques, dont nous venons de parler, insère dans le programme exécutable des références vers les fonctions des bibliothèques, qui seront chargées par ailleurs à l'exécution. L'édition de liens statiques (options -static de gcc) insère le code complet des fonctions dans l'exécutable. Il en résulte un fichier plus gros, mais qui a l'avantage de ne plus dépendre d'autres bibliothèques…

# Chapitre 15

# Exercices

## 15.1 Objectifs

Ces quelques exercices avec corrigés vous permettront de vous familiariser un peu plus avec le langage C. Dans tous ces exercices, on supposera que l'utilisateur ne fait pas d'erreur de saisie.

## 15.2 Jeu de morpion

### 15.2.1 Principe

Il s'agit du morpion classique que l'on pourrait représenter comme ceci :

```
X O .
X O X
O . O
```

Le gagnant est celui qui aligne le premier 3 signes identiques (3 X ou 3 O), sur l'horizontale, la verticale ou la diagonale.

### 15.2.2 Affichage du plateau de jeu

Nous allons définir un tableau pour composer le plateau du morpion et l'initialiser à 0.

Nous dirons qu'une case du plateau de jeu est vide si la case du tableau correspondante contient la valeur 0.



- Écrivez une fonction qui dessine ce plateau.
- Écrivez un programme qui dessine le plateau vide, puis avec les éventuels « O » ou « X ».

### 15.2.3 Saisie des coordonnées



- Écrivez une fonction qui permet la saisie d'une coordonnée.
- Dans un premier temps on ne contrôlera que si la coordonnée est comprise entre 1 et 3.
- Complétez le programme en permettant la saisie des deux coordonnées X et Y.
- Complétez le programme en affichant la grille résultante de la saisie de ces coordonnées (on ne contrôlera pas le fait que la case soit occupée ou non).
- Complétez le programme en testant si la case n'est pas déjà occupée.

### 15.2.4 Alternance des joueurs



Modifiez le programme pour que celui-ci fasse jouer deux joueurs :

- on positionnera la valeur de la case du tableau à 1 pour le joueur 1 et à 2 pour le joueur 2,
- on ne regardera pas qui a gagné dans l'immédiat,
- on affichera `O` pour le joueur 1 et `X` pour le joueur 2.

### 15.2.5 Fin du jeu



- Modifiez le programme pour que celui-ci s'arrête lorsque toutes les cases sont occupées.
- Modifiez le programme pour que celui-ci s'arrête lorsque l'un des deux joueurs a gagné ou lorsque toutes les cases sont occupées.

### 15.2.6 Améliorations possibles

Faites en sorte que l'ordinateur joue en tant que joueur 2 :

- dans un premier temps de manière aléatoire en testant la possibilité de poser son pion (9 cases) ;
- dans un second temps en pondérant les différentes cases du plateau (le centre est la case la plus forte, le coin vient ensuite…)

# 15.3 Jeu de pendu

## 15.3.1 Principe

Le but du jeu est de deviner en moins de 7 essais un mot que seul l'ordinateur connaît. Pour mener à bien votre mission, vous allez proposer une lettre :

– si la lettre est correcte alors, celle-ci s'affiche à sa place dans le mot à deviner ;
– si la lettre est incorrecte, alors, votre nombre d'essais diminue de 1.

Autrement dit :

– lorsqu'une lettre est correcte, le nombre d'essais reste inchangé ;
– lorsqu'une lettre est incorrecte, le nombre d'essais diminue de 1 ;
– lorsque tout le mot a été deviné, vous avez gagné ;
– lorsque le nombre d'essais est à zéro (0), vous avez perdu.

## 15.3.2 Exemple

Supposons que le mot à deviner soit « bonjour ».

Vous proposez la lettre « o », cette dernière se trouve dans le mot, l'ordinateur affiche donc ***o**o****. Si vous proposez ensuite la lettre « u », l'ordinateur affiche : ***o**ou***.

Si vous vous sentez à l'aise, ne lisez pas ce qui suit et essayez de programmer le jeu. Dans le cas contraire, le détail des différentes étapes vous aidera sans doute à réaliser le programme.

## 15.3.3 Un peu d'aide

### 15.3.3.1 Point de départ

Commencez par écrire un programme qui déclare une variable contenant le mot à trouver : `motAtrouver` et une autre contenant une chaine de même longueur remplie avec des * : `motCherche`.

### 15.3.3.2 Algorithme

Un tour de jeu se compose des phases suivantes :

```
Saisie d'un caractère
Initialisation des caractères correspondant dans le mot caché
Si aucun caractère n'a été trouvé
    nombre d'essais -1
Si motAtrouver == motCherche GAGNE
Si nombre d'essais == 0 PERDU
```

### 15.3.4 Améliorations possibles

Libre à vous d'ajouter ce que vous voulez. Voici quelques suggestions :

- lire le mot dans un dictionnaire de mots.
- sauvegarder/recharger une partie en cours.
- gérer les meilleurs scores au moyen d'un fichier.
- dessiner une potence, mettre un peu de couleur.

## 15.4 Balle rebondissante

On souhaite réaliser un programme qui permettrait de faire rebondir une balle sur les bords de l'écran. La balle doit pouvoir rebondir sur un écran de la forme suivante (figure de gauche) :

Afin de faciliter la programmation, on rajoute une couronne d'étoiles qui permettra de simplifier les tests lorsque la balle arrivera sur un bord (figure de droite).

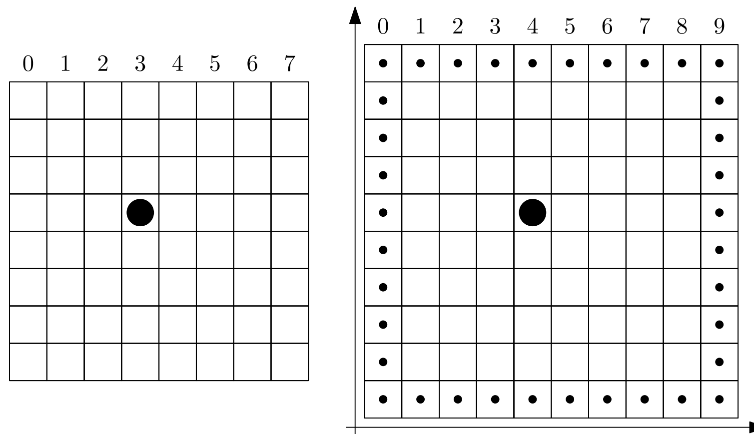



FIGURE 15.1 - Balle rebondissante

La balle, qui est symbolisée par le caractère 'O' démarre à la position `[4;4]`. Elle part suivant le vecteur d'abscisse `+1` et d'ordonnée `+1`.

Lorsqu'elle arrive sur un bord vertical, il suffit d'inverser le sens de déplacement selon l'axe des ordonnées. De même, si la balle arrive sur un bord horizontal, on inverse le sens du déplacement selon l'axe des abcisses.



Pour tester si on arrive sur un bord, il est possible de faire :

```
if (grille[nouvelle_pos_x][nouvelle_pos_y]=='*')
```

Naturellement il est aussi possible de tester les coordonnées `nouvelle_pos_x` et `nouvelle_pos_y` pour savoir si on arrive sur un bord.



Compléter le programme suivant ...

```
#include <stdio.h>
#include <string.h>

/************PROTOTYPES DES FONCTIONS **************************/
/* Initialise la grille de façon à ce qu'elle contienne ce qu'il
  y a à la figure de droite
*/
void init_grille (char grille[][10],int pos_balle_x,int pos_balle_y) ;

/* Affiche le rectangle d'étoiles et la balle (tout ceci en même
  temps et non pas le rectangle puis la balle...)
*/
void affiche_grille (char grille[][10]); /* 10 lignes 10 colonnes */

/* Calcule la nouvelle position de la balle en fonction de
  l'ancienne position de la balle (old_pos_balle_x, old_pos_balle_y)
  et du vecteur de déplacement (deplacement_x, deplacement_y).
*/
void calcule_position_balle (char grille[][10], int *pos_balle_x,
                     int *pos_balle_y, int *deplacement_x, int * →
    ↪ deplacement_y);

/************* IMPLEMENTATION DES FONCTIONS ***************/
void init_grille (char grille[][10], int pos_balle_x,int pos_balle_y)  →
    ↪ {...}

void affiche_grille (char grille[][10]) {...}

void calcule_position_balle (char grille[][10], int *pos_balle_x,
                     int *pos_balle_y,int *deplacement_x,int * →
    ↪ deplacement_y); {...}

int main () {

  int pos_balle_x=4, pos_balle_y=4; /* position balle au départ */
  int deplacement_x=1, deplacement_y=1; /* déplacement balle */

  char grille[10][10] ; /* grille qui contiendra 3 caractères : */
                  /* '*' ou 'O' ou le caractère espace ' '*/

  init_grille (grille, pos_balle_x, pos_balle_y) ;

  while (1) {
   system("clear");

   affiche_grille(grille);

   calcule_position_balle (grille, &pos_balle_x, &pos_balle_y,
                     &deplacement_x, &deplacement_y);

   usleep(500000) ; /* Pause de 500 000 micro secondes donc 1/2 seconde */
  }
}
```

### 15.4.1 Améliorations

Comme on peut le constater, le mouvement de la balle est cyclique, introduisez un déplacement aléatoire de 1 case tous les x tours où x est un nombre aléatoire entre 0 et 9.

## 15.5 Solutions

### 15.5.1 Le morpion

```
#include <stdio.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

void dessine_plateau (int plateau[][3]) {
  int i=0,j=0;

  printf ("\n-------\n");
  for (i=0;i<3;i++) {
    for (j=0;j<3;j++) {
      printf("|");

      switch(plateau[i][j]) {
        case 0:
          printf(" ");
          break;
        case 1:
          printf("O");
          break;
        case 2:
          printf("X");
          break;
      }
    }
    printf ("|\n");
    printf ("-------\n");
  }
}

int fin_jeu (int plateau[][3]) {
  int i=0,j=0;

  for (i=0;i<3;i++) {
    for (j=0;j<3;j++) {
      if (plateau [i][j]==0) {
        return FALSE;
      }
    }
  }
  return TRUE;
}

int saisie_donnee (char *invite) {
  int valeur;

  do {
    printf ("%s", invite);
```

```
    scanf ("%d",&valeur);
  } while (( valeur <1) || (valeur >3));

  return (valeur);
}

int gagne (int plateau[][3]) {
  int i=0;

  // Test sur les lignes
  for ( i=0; i<3; i++) {
    if (( plateau[i][0] >0) && ( plateau[i][0] == plateau[i][1] ) && ( plateau[i →
      ↪ ][1] == plateau[i][2] )) {
      puts ("GAGNE");
      return TRUE;
    }
  }

  // Test sur les colonnes
  for ( i=0; i<3; i++) {
    if (( plateau[0][i] >0) && ( plateau[0][i] == plateau[1][i] ) && ( plateau[1][ →
      ↪ i] == plateau[2][i] )) {
      puts ("GAGNE");
      return TRUE;
    }
  }

  // Test sur les diagonales
  if (( plateau[0][0] >0) && ( plateau[0][0] == plateau[1][1] ) && ( plateau →
      ↪ [1][1] == plateau[2][2] )) {
    puts ("GAGNE");
    return TRUE;
  }

  // Test sur les diagonales
  if (( plateau[0][2] >0) && ( plateau[0][2] == plateau[1][1] ) && ( plateau →
      ↪ [1][1] == plateau[2][0] )) {
    puts ("GAGNE");
    return TRUE;
  }

  return FALSE;
}

void jeu (int plateau[][3], int joueur) {
  int pos_x=0,pos_y=0;
  int correct=FALSE;

  do {
    printf ("Joueur %d\n",joueur);
    pos_x= saisie_donnee ("Ligne : ");
    pos_y= saisie_donnee ("Colonne : ");

    if ( plateau[pos_x-1][pos_y-1]>0 ) {
      printf ("Case occupée !\n");
    } else {
      plateau[pos_x-1][pos_y-1]=joueur;
      correct=TRUE;
    }
  }
  while (! correct);

  dessine_plateau (plateau);
```

```
}

int main () {
  int plateau [3][3];
  int joueur=1;

  // la fonction memset permet d'initialiser chacun
  // des octets d'une zone donnée avec une valeur
  // déterminée (ici: 0)
  memset (plateau, 0, 9*sizeof (int));

  dessine_plateau (plateau);

  do {
    jeu (plateau, joueur);
    if ( joueur==1 ) {
      joueur=2;
    } else {
      joueur=1;
    }
  }
  while (( !gagne (plateau)) && (!fin_jeu (plateau)) );

  return 0;
}
```

### 15.5.2 Le pendu

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

char lireCaractere() {
  char chaine[2];

  gets(chaine);

  return chaine[0];
}

int main() {
  int i=0;
  int coups=7;
  char motAtrouver[]="BONJOUR";
  char lettreSaisie=' ';
  int lettre_trouvee=FALSE;
  char gagne=FALSE;

  char* motCherche;
  motCherche=malloc (sizeof (motAtrouver));
  memset (motCherche,'*',sizeof (motAtrouver));
  motCherche[sizeof (motAtrouver)-1]=0;

  printf("Jeu de pendu \n");

  do {
    // Aucune lettre trouvée
```

```
    lettre_trouvee=FALSE;

    // Saisie d'une lettre et mise en majuscule
    printf("\nVotre lettre : ");
    lettreSaisie=lireCaractere();

    // Comparaison avec le mot secret
    for(i=0; i<strlen (motAtrouver); i++) {

      if(lettreSaisie==motAtrouver[i]) {
       motCherche[i]=lettreSaisie;
       lettre_trouvee=TRUE;
      }
    }

    printf("%s", motCherche); //on affiche le mot cache
    if (!lettre_trouvee) {
      coups--;
    }
    printf("\nIl vous reste %d coups.\n ", coups );

    gagne=! strcmp(motAtrouver, motCherche);
  }
  while(!gagne && coups>0);

  if ( gagne )
    puts ("GAGNE");
  else
    puts ("PERDU");

  getchar();

  free (motCherche);

  return 0;
}
```

### 15.5.3 Balle rebondissante

```
#include <stdio.h>
#include <string.h>

/************PROTOTYPES DES FONCTIONS **************************/
/* Initialise la grille de façon à ce qu'elle contienne ce qu'il
   y a à la figure de droite
*/
void init_grille (char grille[][10],int pos_balle_x,int pos_balle_y) ;

/* Affiche le rectangle d'étoiles et la balle (tout ceci en même
   temps et non pas le rectangle puis la balle...)
*/
void affiche_grille (char grille[][10]); /* 10 lignes 10 colonnes */

/* Calcule la nouvelle position de la balle en fonction de
   l'ancienne position de la balle (old_pos_balle_x, old_pos_balle_y)
   et du vecteur de déplacement (deplacement_x, deplacement_y).
*/
void calcule_position_balle (char grille[][10], int *pos_balle_x,
                     int *pos_balle_y, int *deplacement_x, int *deplacement_y);
```

```
/***************** IMPLEMENTATION *****************************/

void init_grille(char grille[][10],int pos_balle_x,int pos_balle_y){
  int ligne, colonne;

  memset (grille,' ',100);

  for (colonne=0; colonne <10; colonne++) {
    grille [0][colonne]='*';
    grille [9][colonne]='*';
  }

  for (ligne=0; ligne<10; ligne++) {
    grille [ligne][0]='*';
    grille [ligne][9]='*';
  }

  grille [pos_balle_x][pos_balle_y]='O';
}

void affiche_grille (char grille[][10]) {
  int ligne, colonne;
  for (ligne=0; ligne<10; ligne++ ) {
    for (colonne=0; colonne <10; colonne++) {
      printf ("%c",grille[ligne][colonne]);
    }
    printf ("\n");
  }
}

void calcule_position_balle (char grille[][10], int *pos_balle_x,
                      int *pos_balle_y,int *deplacement_x,int *deplacement_y) {

  int theo_pos_x=0;
  int theo_pos_y=0;

  // On efface l'ancienne balle
  grille[*pos_balle_x][*pos_balle_y]=' ';

  printf ("Position actuelle : %d / %d\n",*pos_balle_x,*pos_balle_y);
  printf ("Déplacement : %d / %d\n",*deplacement_x,*deplacement_y);

  // On calcule la future position théorique de la balle
  theo_pos_x = *pos_balle_x + *deplacement_x;
  theo_pos_y = *pos_balle_y + *deplacement_y;

  // En fonction de la position théorique de la balle
  // on modifie les vecteurs de déplacement
  if (grille[theo_pos_x][theo_pos_y]=='*') {
    // Si on tape sur l'axe vertical
    if (( theo_pos_x == 0 ) || ( theo_pos_x == 9 ))
      *deplacement_x = - *deplacement_x;

    // Si on tape sur l'axe horizontal
    if (( theo_pos_y == 0 ) || ( theo_pos_y == 9 ))
      *deplacement_y = - *deplacement_y;
  }

  // On calcule la nouvelle position de la balle
  *pos_balle_x += *deplacement_x;
  *pos_balle_y += *deplacement_y;

  printf ("Nouvelle Pos : %d/%d\n",*pos_balle_x,*pos_balle_y);
```

```
  // On met la balle dans la grille
  grille[*pos_balle_x][*pos_balle_y]='O';
}

int main () {

  int pos_balle_x=4, pos_balle_y=4; /* position balle au départ */
  int deplacement_x=1, deplacement_y=1; /* déplacement balle */

  char grille[10][10] ; /* grille qui contiendra 3 caractères : */
                   /* '*' ou 'O' ou le caractère espace ' '*/

  init_grille (grille, pos_balle_x, pos_balle_y) ;

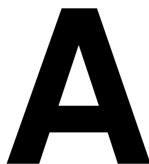
  while (1) {
   system("clear");

   affiche_grille(grille);

   calcule_position_balle (grille, &pos_balle_x, &pos_balle_y, &deplacement_x, & →
    ↪ deplacement_y);

   usleep(500000) ; /* Pause de 500 000 micro secondes donc 1/2 seconde */
  }
}
```

Annexe



# Petit extrait de la table Ascii

```
  ... ...
  32:      33: !    34: "    35: #    36: $    37: %    38:      39: '
  40: (    41: )    42: *    43: +    44: ,    45: -    46: .    47: /
  48: 0    49: 1    50: 2    51: 3    52: 4    53: 5    54: 6    55: 7
  56: 8    57: 9    58: :    59: ;    60:      61: =    62:      63: ?
  64: @    65: A    66: B    67: C    68: D    69: E    70: F    71: G
  72: H    73: I    74: J    75: K    76: L    77: M    78: N    79: O
  80: P    81: Q    82: R    83: S    84: T    85: U    86: V    87: W
  88: X    89: Y    90: Z    91: [    92: \    93: ]    94: ^    95: _
  96: `    97: a    98: b    99: c   100: d   101: e   102: f   103: g
 104: h   105: i   106: j   107: k   108: l   109: m   110: n   111: o

... ...
```

# Annexe B

# Bon à savoir

| Type de donnée | Signification | Taille (en octets) | Plage de valeurs |
|---|---|---|---|
| **`char`** | caractère | 1 | $-128$ à 127 |
| **`int`** | entier | 4 | $-2147483648$ à 2147483647 |
| **`float`** | flottant (réel) | 4 | $3.4 * 10^{-38}$ à $3.4 * 10^{38}$ |
| **`double`** | flottant double | 8 | $1.7 * 10^{-4932}$ à $1.7 * 10^{4932}$ |

Table B.1 - Les types numériques les plus utiles

**Les tableaux suivants sont à connaître par coeur !**

| Format | Conversion en |
|---|---|
| %d | **int** |
| %f | **float** |
| %f | **double** |
| %c | **char** |
| %s | **char*** |

TABLE B.2 - Les formats les plus utiles pour printf

| Format | Conversion en |
|---|---|
| %d | **int** |
| %f | **float** |
| %lf | **double** |
| %c | **char** |
| %s | **char*** |

TABLE B.3 - Les formats les plus utiles pour scanf

Bien des erreurs pourraient être évitées en maîtrisant ce tableau :

| Erreur | Version correcte |
|---|---|
| `if (i=0)` | `if (i==0)` |
| `scanf ("%d",n);` | `scanf ("%d",&n);` |
| `scanf ("%s",s)` est équivalent à `gets(s)` | `gets(s)` permet de lire une phrase complète<br>(jusqu'à l'appui sur ENTREE ) |
| `if (a & b)` | `if (a && b)` |
| oublier d'initialiser une variable | |
| `tab[i,j]` | `tab[i][j]` |
| les bornes d'un tableau de `N` cases varient entre `1` et `N` | les bornes d'un tableau de `N` cases varient entre `0` et `N-1` |
| `char c;`<br><br>`printf ("%s",c);` | `char c;`<br><br>`printf ("%c",c);` |
| `char * s;`<br><br>`printf ("%c",c);` | `char * s;`<br><br>`printf ("%s",c);` |
| `char chaine[3]="12345";` | `char chaine[6]="12345";`<br><br>ou mieux :<br><br>`char chaine[]="12345";` |
| `char chaine[]='12345';` | `char chaine[]="12345";` |
| si vous utilisez des fonctions mathématiques telles que `sqrt`… | compilez le programme avec `-lm`<br><br>`gcc -O essai essai.c -lm` |

TABLE B.4 - Douze erreurs parmi les plus classiques en langage C

# Table des matières

# Table des figures

# Liste des tableaux

* * *

# Achevé d'imprimer

Achevé d'imprimer en 2010 en France pour le compte d'InLibroVeritas

ISBN : 978-2-35922-030-8
Dépôt légal : 2e semestre 2010




Cet ouvrage se propose de vous faire découvrir par la pratique toutes les bases essentielles du Langage C. Destiné aux grands débutants, fruit de l'expérience pédagogique d'Eric Berthomier et Daniel Schang, l'ouvrage n'est pourtant pas un manuel comme les autres. Son intérêt est d'être parcouru de A à Z en suivant l'adage : « j'apprends le code en écrivant le code ».
Testée par de nombreux étudiants qui n'avaient aucune connaissance préalable du Langage C, la méthode développée par les auteurs permet d'effectuer un apprentissage complet en une durée de 20 à 30 heures de travail.

**Eric Berthomier**
Ingénieur en Informatique, Eric Berthomier a débuté par le développement d'applications systèmes en C / C++ / Assembleur. Investi dans le Libre, il réalise alors des missions de développement, d'administration systèmes et réseaux et de formation.
Depuis 2005, il travaille pour un Ministère où Linux et la sécurité sont ses compagnons de tous les jours. Le C (ou l'assembleur) est pour lui indispensable à la compréhension d'un système d'exploitation.



**Daniel Schang**
Docteur en informatique, Daniel Schang est enseignant-chercheur au sein du Groupe Eseo où il a acquis une longue et riche expérience de l'enseignement des langages informatiques. À l'écoute de ses élèves, c'est pour eux qu'il a pris contact avec Eric Berthomier afin de réécrire ce livre.



*La route est longue mais la voie est libre*
**Framasoft**

**Cet ouvrage s'inscrit dans la collection de livres libres du réseau Framasoft**